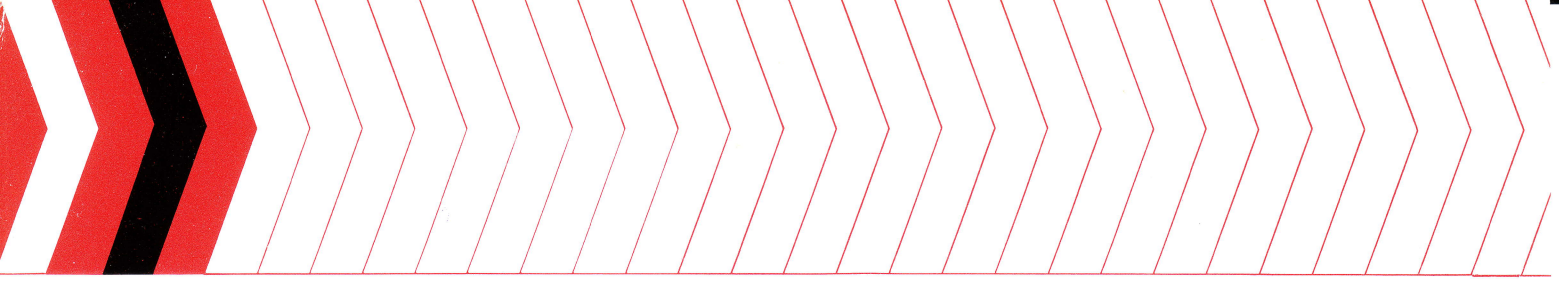
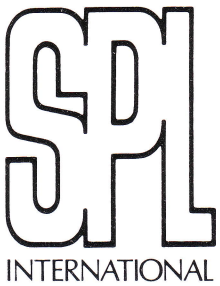
Introduction to RTL/2





RTL/2 high-level computer language
was designed and developed by
Imperial Chemical Industries Limited.
RTL/2 software for use in conjunction with
Digital Equipment Corporation PDP-11,
IBM System 360/370 and ICL System 4 computers
is marketed in the UK and Western Europe by
SPL International
12-14 Windmill Street, London W1P 1HF.

Introduction

RTL/2 is a high-level programming language. It is designed for use in real-time computing and is especially suited for the programming of on-line data collection, communication and control systems. The language is independent of particular types of hardware and is practical to use on small computers. Advantages in use include lower programming costs, faster implementation and improved software reliability.

RTL/2 (Real Time Language) was developed in the Corporate Laboratory of Imperial Chemical Industries Limited, England. The project was initiated in 1969 following some earlier exploratory work. The objective was to develop an improved method of programming real-time computer applications as an essential step towards full exploitation of low cost hardware. To be effective the method had to be capable of generating compact and highly reliable multi-task software on a wide range of computers.

A prototype language RTL/1 was implemented in 1970 and was employed successfully in two major process control applications. RTL/2 was defined in 1971 and its first compilers were made available early in 1972. Since then RTL/2 has been in regular use by ICI computer application teams and has achieved its full design objectives. The basic language definition is therefore frozen and there are no plans for further extension or modification.

The range of supporting software will continue to develop and includes :

* compilers and linkers for translating RTL/2 programs into code for different types of computer.
* system standards and software to aid the development and running of RTL/2 programs.
* RTL/2 library programs.

Information on RTL/2 documentation and software items currently available is included in the pocket of this manual.

A brief description of the RTL/2 language and its application follows.

Key Features

RTL/2 is specifically designed for use in a real-time environment. It can be used for conventional programming, but incorporates basic features that are particularly important for the programming of on-line computer applications.

PURPOSE BUILT

ability to express *multi-task* programs in which activities proceed in parallel
All code is re-entrant and tasks may have identifiable private data areas ; these, together with high-level supervisor calls, provide the basis for efficient task control and interaction. They simplify the planning and writing of software for computer systems that have to respond automatically to external events.

RELIABLE

highly developed *safeguards* against programming error
The language structure enables stringent checks to be applied automatically during compiling and linking of programs ; it also provides for efficient run-time checking and error recovery. These are essential for real-time applications in which errors may cause irretrievable loss of data or product and are difficult to reproduce for diagnostic purposes.

EFFICIENT

ability to generate *compact code* with low run-time overheads
Variables and data structures are designed for economic handling of real-time data and communications ; they include bytes, fraction variables and record structures. Reference, or "pointer", variables enable further structures such as lists and queues to be created and lead to efficient use of modern hardware for data accessing and procedure calls.

FLEXIBLE

a highly *modular* program structure
Interfaces between modules are explicitly defined and can be checked for valid correspondence. This aids overall software management and safe modification of on-line programs.

PRACTICAL

ease of application
RTL/2 is a well-defined language ; it can be taught in a few days and the absence of ad hoc rules and restrictions makes it easy to remember. The notation is conventional and can be typed on standard keyboard equipment. The 'size' of the language makes it feasible to compile on a typical 16K computer.

The above combination of features reflects the fact that RTL/2 has been specifically engineered for real-time programming.

RTL/2 IS NOT

simply a variant or subset of an existing data-processing language, for use on small computers.

a set of extensions to an existing language, implemented by means of an operating system that has to be provided on every computer.

specific to process control (although many of its early applications were in this field).

dependent on a particular type of computer or standard software system.

an unproven language proposal yet to be implemented.

Applications

RTL/2 comes into its own on real-time applications, especially (but not exclusively) where small computers are involved. This field typically includes systems for communications, commercial data entry, process control and factory automation, traffic control, and the monitoring and analysis of research data. It also includes the wide range of special systems produced by OEM suppliers.

Within this field, RTL/2 may be used at various levels :

SPECIAL SYSTEMS

for programming *special systems*
RTL/2 programs need very little additional software at run-time. Thus minicomputer applications not requiring a full range of system facilities need not incur the overheads ; they can be programmed directly in RTL/2. The software development is done on a larger and more convenient off-line machine.

OPERATING SYSTEMS

for writing *standard operating systems*
Task supervisors, standard input/output drivers and utilities can be built up in a modular fashion using the minimum of assembly code ; real-time operating systems are in use for which 95% of the code has been generated from RTL/2. Such systems are easier to understand and adapt to special requirements – for example, addition of an interface to a special I/O device or to another computer. To the extent that they are written in RTL/2, they are also less machine-dependent.

APPLICATION PROGRAMS

for *application programming*
Real-time application programs will generally define a set of tasks to be scheduled and serviced by an operating system. The latter may be a system written in RTL/2 or a supplier's standard system to which RTL/2 has been interfaced. Application programs communicate with the system through RTL/2 procedures and supervisor calls ; the recommended RTL/2 standards for I/O and error recovery apply to both single-task and multi-task systems.

STANDARD PACKAGES

for writing *standard packages*
These include, for example, standard programs for process control and sequencing, or for implementing conversational systems.

The use of RTL/2 is not confined to real-time systems or small computers. It may also be used :

APPLICATION PROGRAMMING

for *general application programming*
Although RTL/2 is not designed for the same purpose as FORTRAN or ALGOL its algorithmic facilities compare favourably with such languages.

SOFTWARE AIDS

for writing *system utilities and various software aids*
The data structures and character handling in RTL/2 enable it to be considered as an alternative to assembly code for system programming. The re-entrancy and other real-time features are also relevant.

RTL/2 compilers are themselves written in RTL/2.

Practical Benefits

RTL/2 offers the usual advantages of a high-level language over assembly code : cheaper, quicker and better documented programming plus the ability to run software on different computers. It is of special benefit in on-line applications where smooth commissioning and reliability are the critical factors ; for example, in the control of a continuous industrial process. Experience in use of RTL/2 has demonstrated the following benefits in the stages of such a project :

PLANNING

The re-entrancy of RTL/2 code and natural task structure allow design effort to be concentrated on the actual computer tasks and their organisation. Moreover, where the operating system is also written in RTL/2, its ease of adaptation permits a more flexible approach to the overall design. This applies particularly to small systems.

WRITING, COMPILING & ASSEMBLY

A several-fold increase in productivity, depending on the nature of the job and software tools available. This is due partly to the modularity of RTL/2, which enables the work to be tackled in a well organised manner, and partly to the range of language facilities, expressly designed to minimise the need for assembly code in real-time programming.

COMMISSIONING

The integrity features built into RTL/2 enable the programs to be thoroughly checked during both compiling and linking : thus programs reaching the commissioning stage should be free from all but fundamental logical errors. Moreoever, if the programs have been compiled with run-time error monitoring provided by the language, there is a good chance that such errors will be trapped and quickly diagnosed. Experience has shown that the integrity features of RTL/2 are a valuable asset during the commissioning of on-line computer installations, especially if the hardware is late or being troublesome.

MAINTENANCE & POST-DEVELOPMENT

Maintenance is assisted by the clearer documentation of programs and continued use of run-time error detection and recovery to trap unforeseen error situations. The modular nature and integrity of RTL/2 software makes it easier and safer to modify as requirements change, especially if the work has to be undertaken by someone other than the original programmer.

```
MODE CTCELL(ARRAY(8)BYTE NAME,INT COUNT);

MODE RTCELL(INT RNAME,RCOUNT);

MODE LKCELL(INT CB,REF LKCELL LAST,REF PROCHD THISPROC,BYTE RA1,RA2,
   ARRAY(0)INT LOCALS);

MODE STK(INT STKLNGTH,REF LKCELL BASECELL,INT TOPSTACK,BYTE MS1,MS2,
   ARRAY(16)INT GREGS,REAL FPR0,INT PCOUNTER,LABEL SVERL,INT SVERN,
   PROC(INT) SVERP,PROC()BYTE SVIN,PROC(BYTE) SVOUT,BYTE SVTERMCH,
   SVIOFLAG,SVSTSFLG,GOTOFLAG,INT STKSPARE);

%%%%%%%% RRDBG %%%%%%%%

ENT PROC RRDBG(REF STK X);
   INT ERNO:=X.SVERN;
   REF LKCELL CELL;

   SET();
   CELL:=X.BASECELL;
   TWRT("#NL#RTL ERROR NUMBER ");
   IWRT(ERNO);
   TWRT(" ON LINE ");
   IWRTS(X.GREGS(7),3);      % GR 6 HOLDS LINE NUMBER %
   TWRT("#NL#          IN PROC ");
   NAMELOC(CELL);
   WHILE CELL.LAST:#:X.BASECELL DO
      CELL:=CELL.LAST;
      TWRT("#NL#CALLED FROM PROC ");
      NAMELOC(CELL);
   REP;

   IF ERNO>=10000 THEN
      TWRT("#NL#PROGRAM COUNTER = ");
      XWRT(X.PCOUNTER);
   END;

   TWRT("#NL#REGISTERS 0-7  ");
   FOR I:=1 TO 8 DO
      SPS(2);
      XWRT(X.GREGS(I));
   REP;

   TWRT("#NL(2)#RETROTRACE#NL#  ");
   FOR I:=15 BY -1 TO 0 DO
      BLOCK
         REF RTCELL RTC:=RTABLE(((I+RPTR SRL 3) LAND 15)+1);
         INT RN:=RTC.RNAME;

         IF RN#0 THEN
            TWRT(CTABLE(RN SRL 4).NAME);
            SPS(2);
            IWRT(RTC.RCOUNT-RN+1);
            TWRT("#NL#  ");
         END;
      ENDBLOCK;
   REP;
   RRCNT();
ENDPROC;
```

Example of RTL/2 system programming

Use of RTL/2

The steps in developing an RTL/2 program follow the conventional pattern. The program is typed or input to a computer on standard equipment and then translated into code using an RTL/2 compiler. A large program may be written and compiled as several independent modules which are then linked together. The language definition enables a well-written compiler to thoroughly check that programs conform exactly to the rules of RTL/2 : explicit cross–references enable further checks to be made during linking of modules.

In the case of a real-time program the first step in the design is to clearly identify the various run-time tasks and their interactions. The following example illustrates how the RTL/2 language structure assists at this planning stage :

A computer system for industrial control might have the following specification :

* scan a series of analogue and digital inputs at regular intervals.
* check each point for alarm conditions, printing or displaying alarm warnings.
* compute control actions (direct or supervisory) based on some of the scanned points.
* apply sequences of control actions on certain sections of the plant, spanning periods of minutes or hours.
* display information on demand and read plant control data from an operators panel.

This specification implies a set of parallel activities within the computer, some at regular but differing time intervals and some irregular. The activities require access to common areas of data describing the current state of the plant and control settings. Each activity also requires a private work space in which to hold temporary data arising during calculations or indicating its current status. Finally each activity requires its defining code, or program, and (for storage economy or other reasons) some programs may need to be shared as common routines.

These requirements are reflected in the program structure of RTL/2. All RTL/2 software is built-up of units called 'bricks', of which there are three basic types : procedure bricks, stack bricks, and data bricks.

A **procedure brick** is a set of RTL/2 statements. All procedures are defined in RTL/2 to be re-entrant, meaning that the code is never changed during execution and all temporary data is stored elsewhere. Thus a procedure may serve several activities, or **tasks,** at the same time ; for example, a procedure defining a sequence of control actions may be applied to two plant streams simultaneously, or a procedure required by a high-priority task may be used without waiting for a lower-priority task to finish with it. This distinction between a procedure – a passive set of instructions – and a task – the execution of a procedure for a particular purpose – is fundamental in RTL/2.

The parallel use of procedures is accomplished by assigning a **stack brick** or temporary work area, to each active task. In this stack, the work-space needed by procedures used by the task is kept separate from the workspace for another task using the same procedures. Access to data in each stack is looked after automatically and change of task simply means change to another working stack. The ability to assign and refer to stacks explicitly (and hence to their associated tasks) is the other important ingredient in RTL/2 which makes multi-task programming possible.

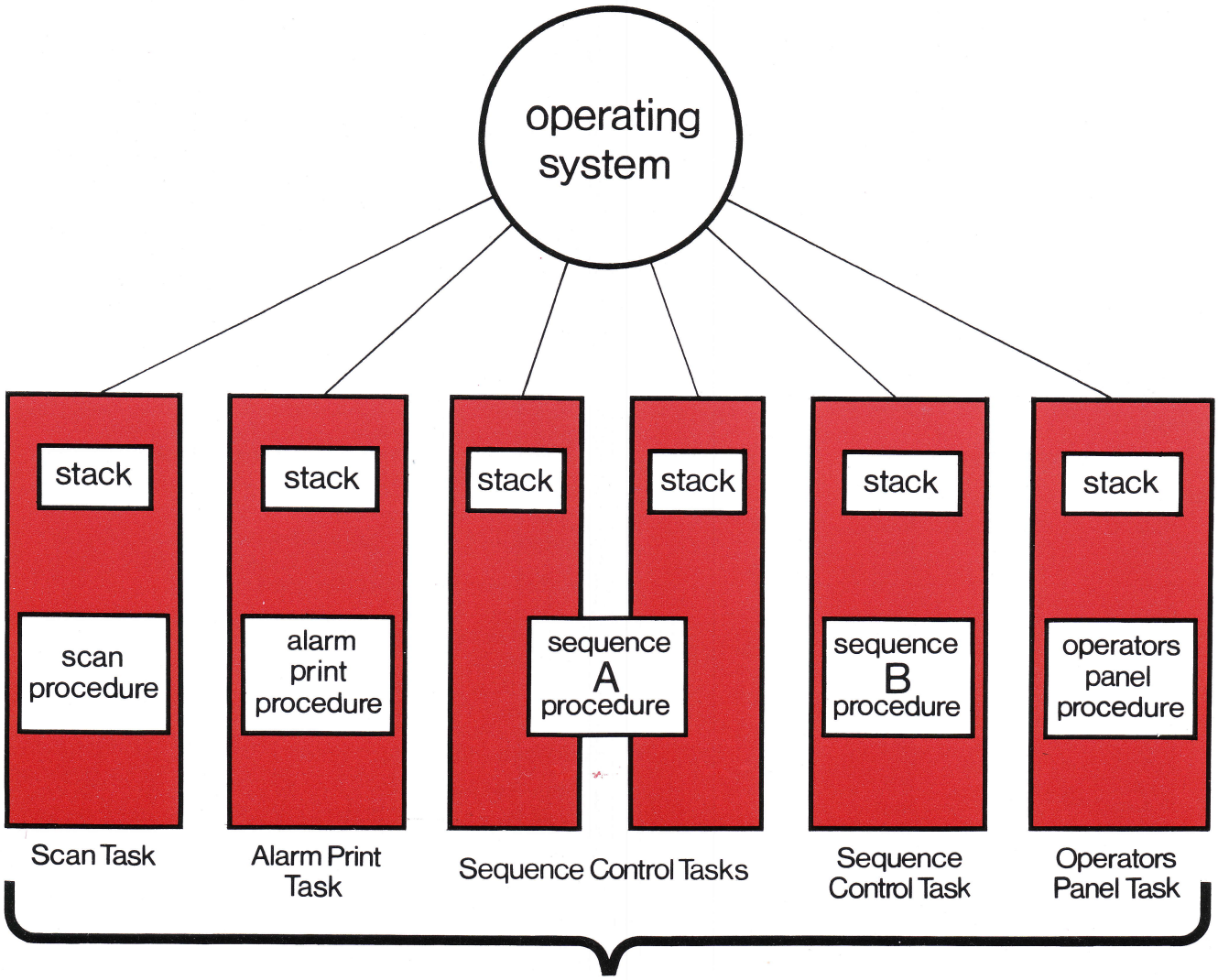
Finally, RTL/2 allows the programmer to create any number of **data bricks** or named common data areas. These may be used to store data of interest to several tasks or data which outlives the duration of individual tasks.

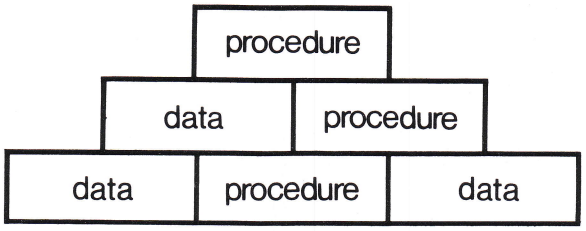
Returning to the industrial control example, it is now possible to see how to plan the programming in RTL/2. The first step is to identify the tasks :

* SCAN TASK : Scans points, checks alarm states, computes and outputs control actions. Runs regularly and frequently.
* ALARM TASK : Prints out alarm messages on demand from scan task (this might be part of the scan task if the time scale for printing matches that for scanning).
* SEQUENCE TASKS : One for each section of the plant which is wholly or partially time-independent. Takes the plant through a sequence of operations, monitoring its progress. Called by operator or from other tasks.

  For the purpose of this example it is assumed that two identical sections of the plant require the same control sequence A and a third section requires a different sequence B.
* OPERATORS PANEL TASK : Responds to operator demands (via interrupts) and inputs or outputs data as requested.

It will be seen that activities have been grouped into tasks each of which has its own natural time scale.





further procedure and data bricks used by tasks on a private, shared or global basis

Following this step, the programmer can now plan the procedures and data bricks which he will need for these tasks. The total software structure, including one RTL/2 stack per task, will have a layout resembling the diagram above.

The procedure, stack and data bricks have now to be specified in RTL/2, compiled and checked. At this stage the bricks may be dealt with in groups or modules convenient for teamwork and general program management. The compiled modules are finally linked together to form the complete system with checks applied for correct matching of references between modules.

In practice some of the software required by the user will already be available in the form of standard packages. In the type of application illustrated above he would generally expect to use a standard operating system to control and supervise the tasks, possibly itself programmed in RTL/2.

Summary of RTL/2 language features

CHARACTER SET (ISO 7)

A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
" % ' ( ) * + , — . / : ; < = >
£ $ (*interchangeable*)
& ? @
HT (*horizontal tab*)
LF (*line feed*)
SP (*space*)

RTL/2 ITEMS

Item is terminated by any character not part of it.

Names

name ::= letter [letter|digit]...

Reserved keywords

ABS	ENDPROC	MODE	SLA
AND	ENT	NEV	SLL
ARRAY	EXT	NOT	SRA
BIN	FOR	OCT	SRL
BLOCK	FRAC	OF	STACK
BY	GOTO	OPTION	SVC
BYTE	HEX	OR	SWITCH
CODE	IF	PROC	THEN
DATA	INT	REAL	TITLE
DO	LABEL	REF	TO
ELSE	LAND	REP	VAL
ELSEIF	LENGTH	RETURN	WHILE
END	LET	RTL	
ENDBLOCK	LOR	SHA	
ENDDATA	MOD	SHL	

Numbers

integer	real	fraction
456	3.47	0.493B1
BIN 101	0.1E7	17.6B—5
OCT 37	76E—4	1E10B—36
HEX AF2	1E+2	—1E—1B+1
'A'	0.0	0.0B0

actual space and all RTL printable characters except " # £ $ allowed between ' '

Strings

"THIS IS A STRING"
"#NL(2)#TABLE HEADING" assumes LET NL=10;
"**#9,9#MESSAGE#10#"

(or £ or $) enclose numeric insert
newline not allowed in strings but concatenated strings treated as one

"THIS IS THE FIRST PART"
"AND THIS IS THE REST"

Comment

%THIS IS A COMMENT%

Option

OPTION(1) BC, CM, TR;
OPTION(2);

option items always available are

BC bound checks on access
BS bound checks on storage
NW no warning messages
NS no scope warning messages

others depend on implementation.

Let

Text replacement

LET NL=10;%NEWLINE IS ISO7 10%
LET MASK=OCT 1000;
LET RAB=REF ARRAY BYTE;
LET NR=5;%NUMBER OF REACTORS%
LET ATMOS=14.7;%PRESSURE IN PSI%

DECLARATIONS

scalars : local to procedure or global in databrick
arrays and records : only global in databrick

General syntax is

type item [, item]...

where type describes characteristics of identifiers being declared and item is either an identifier or a group of identifiers with initial value separated by :=

Scalar Declarations

primitive modes : BYTE, INT, FRAC, REAL,
LABEL, PROC descriptor, STACK
also REF scalar, REF array, REF record.

Descriptor of PROC variable lists parameter modes in brackets and followed by result mode – same as external description of PROC bricks.

without initial values thus :

INT I, J, COUNTER;
REAL HEIGHT;
LABEL RESTART;

with initial values thus :

INT K:=—310, R:=OCT 37;
BYTE TERMINATOR:='*', FLAG:=1;
REAL PRESSURE:=0.0;
PROC(REAL)REAL FN:=SIN; %SIN IS PROC BRICK%
REF INT II:=K;
REF ARRAY BYTE MM:="MESSAGE";

initial values for local declarations may be any valid expression; in data brick must be :

BYTE/INT/FRAC/REAL	: suitable number
PROC/STACK	: brick name
LABEL	: not possible
REF scalar	: variable name
REF array	: array name/string
REF record	: record name

all variables except plain and LABEL must be initialised in application language.

Array Declarations

Can have arrays of scalars and arrays of records but not arrays of arrays ; multidimensional arrays are actually arrays of REF arrays. Lower bounds always 1. Upper bound (=length) may be zero.

ARRAY(3)INT AI, AJ;
ARRAY(7)REF ARRAY REAL RPTRS;
ARRAY (5, 10) BYTE TABLE;

Initial values are denoted by list of element values in brackets – repetition factors (also in brackets) may be applied. In case of array of bytes may be denoted by string. Structure nested in multidimensional case.

```
ARRAY(12)INT DAYSINMONTH:=(31, 28, 31, 30, 31,
                           30, 31 (2), 30, 31,
                           30, 31);
ARRAY (2, 2) REAL UNITMAT:= ((1.0, 0.0), (0.0, 1.0));
ARRAY(16) BYTE HEXDIG:="0123456789ABCDEF";
ARRAY(5) REF ARRAY BYTE STEPS:=("SLOW" (2),
                                "QUICK" (2),
                                "SLOW");
```

Record Declarations

Components of record may be scalars or arrays but not records or arrays of records. Component may be REF record. Shape of record described by MODE definition at brick level.

MODE name (type namelist [, type namelist] . . .) ;

```
MODE COMPLEX (REAL RL, IM) ;
MODE LIST (INT HD, REF LIST TL) ;
```

Actual records and arrays of records declared as for scalars. Initial values denoted by list of component values in brackets – but no repetition factors.

```
COMPLEX I := (0.0, 1.0) ;
ARRAY (100) LIST MAIN := ((0, DUMMY) (100)) ;
LIST DUMMY := (0, DUMMY) ;
```

BRICKS

ENT before brick makes accessible from outside.
EXT (or SVC, but no SVC stack !) before description of brick indicates is outside.

Data Declarations

[ENT] DATA name ; databody ; ENDDATA ;
Databody is collection of declarations

```
ENT DATA MISC ;
  INT COUNTER ;
  REAL TEMP, PRESS ;
  ARRAY(5) BYTE MARK := "PQLXV" ;
ENDDATA ;
```

External description omits initial values.

Stack Declarations

[ENT] STACK name length ;

```
ENT STACK MYSTACK 500 ;
```

External description omits length.

```
EXT STACK MYSTACK ;
```

Proc Declarations

ENT PROC name (paradescription) [resultmode] ;
blockbody ; ENDPROC ;

Blockbody describes action ; paradescription describes and names parameters – brackets always present even if no parameters ; result mode describes result if procedure is to be a function.

```
PROC SUM (REF ARRAY REAL A) REAL ;
  REAL T := 0.0 ;
  FOR I := 1 TO LENGTH A DO T := T + A (I) REP ;
  RETURN (T) ;
ENDPROC ;

ENT PROC STOP () ;
L : GOTO L ;
ENDPROC ;

PROC MP (INT I, J, REAL R, LABEL L) BYTE ;
.
.
ENDPROC ;
```

External description has name last and no names in parameter list.

```
EXT PROC (REF ARRAY REAL) REAL SUM ;
EXT PROC () STOP ;
EXT PROC (INT, INT, REAL, LABEL) BYTE MP ;
```

Stack and proc descriptions may describe several bricks.

```
EXT PROC (REAL) REAL SIN, COS, TAN, SQRT ;
```

EXPRESSIONS

Basic constituents are constants (numbers or literal names), variables, function calls and conditional expressions. Dyadic and monadic operators may be applied with brackets overriding normal precedences.

Variables

Array elements denoted by appending subscript expressions in brackets to array or ref array.

Record components denoted by appending dot (.) and component name to record or ref record.

```
DAYSINMONTH (7)
UNITMAT (I, J+1)
STEPS (I, J)   or   STEPS (I)(J)
DUMMY.TL
MAIN(3).TL.HD
```

Function Calls

Procedure name followed by actual parameter list in brackets. Brackets present even if no parameter. Parameters are any expression of appropriate mode.

```
Z := LOG (X+Y) ;
T := TIME () ;
AJ(3) := F (J, K, L—3) ;
```

Conditional Expressions

IF condition THEN expression
[ELSEIF condition THEN expression] . . .
ELSE expression END

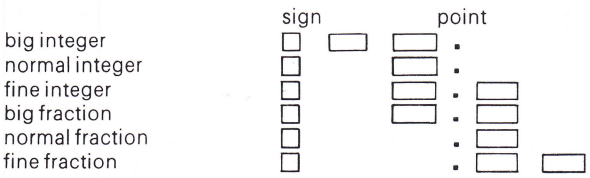
Condition built out of comparisons with AND and OR ; AND is more tightly binding and comparisons evaluated from left to right until condition determined. Brackets cannot be used to override the precedence of AND/OR.

Comparison operators :

= ≠	on 7 primitive modes
< <= > >=	on INT, FRAC, REAL
:=: :≠:	on 9 REF modes

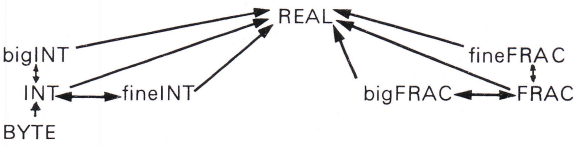
```
I := IF X=0 THEN 3 ELSE I+J END ;
B := IF X=Y AND P<0 THEN 'A' ELSE 0 END ;
FLAG := IF MAIN (I).TL := :D THEN 0 ELSE 1 END ;
RWRT (IF P=1 THEN 0.0 ELSEIF P>1 THEN 3.7 ELSE
                                        2.4 END) ;
TWRT (IF L=0 THEN "THIS" ELSE "THAT" END) ;
```

Double Length



NB fine integer same form as big fraction.

Automatic Conversion



big ⟶ normal may cause overflow
fine ⟶ normal rounded

Conversion is automatic where no loss of information can arise ; in other cases it must be forced by BYTE, INT, FRAC operators

Monadic Operators

	operand	result
+, ABS	BYTE, INT, FRAC, REAL	as operand
—	FRAC, INT, REAL	as operand
NOT	INT	INT
INT	bigFRAC	fineINT
	INT, REAL	INT
FRAC	fineINT	bigFRAC
	FRAC, REAL	FRAC
BYTE	BYTE, INT, REAL	BYTE
REAL	BYTE, INT, FRAC, REAL	REAL
LENGTH	array	INT

REAL ⟶ INT and REAL ⟶ FRAC are rounded

Dyadic Operators

	precedence	operands		result
SLL, SRL, SHL	6	INT	INT	INT
SLA, SRA, SHA	6	see below	INT	see below
*	5	INT	INT	bigINT
		INT	FRAC	bigFRAC
		FRAC	INT	bigFRAC
		FRAC	FRAC	fineFRAC
		REAL	REAL	REAL
:/	5	bigINT	INT	INT
		fineINT	FRAC	INT
		bigFRAC	FRAC	INT
//	5	fineINT	INT	FRAC
		bigFRAC	INT	FRAC
		fineFRAC	FRAC	FRAC
/	5	REAL	REAL	REAL
MOD	5	bigINT	INT	INT
		fineINT	FRAC	FRAC
		bigFRAC	FRAC	FRAC
LAND	4	BYTE	BYTE	BYTE
		INT	INT	INT
LOR	3	BYTE	BYTE	BYTE
		INT	INT	INT
NEV	2	BYTE	BYTE	BYTE
		INT	INT	INT
+—	1	INT	INT	INT
		FRAC	FRAC	FRAC
		REAL	REAL	REAL

Arithmetic Shifts – Result Types

First operand	SLA	SHA	SRA
big	big	big	big
normal	big	normal	fine
fine	fine	fine	fine

STATEMENTS

Statements may be labelled by prefixing by identifier and colon. May be several labels.

```
L:A:=B;
M1:M2:RETURN;
```

Block

BLOCK blockbody ENDBLOCK
where blockbody ::= [simpledec;]... sequence

```
BLOCK INT S:=0;
  FOR I:=1 TO N DO S:=S+Q(I) REP;
  IWRT(S);
ENDBLOCK;
```

Assignment Statement

destination := [destination :=]... expression
Each destination consists of a variable possibly preceded by VAL

```
I:=J:=0;
A(I):=J+1;
MAIN(2).TL:=DUMMY;
VAL II:=II+1;
MM:=IF FLAG#0 THEN '' ''ELSE'' HALT'' END;
```

Goto Statement

GOTO label-expression

```
GOTO FINISH;
```

Switch Statement

SWITCH expression OF labellist
Labellist must consist of local literal labels. If expression out of range then no jump occurs.

```
SWITCH K OF P1, P2, P3, P4, P5;
TWRT (''K OUT OF RANGE'');
```

Conditional Statement

IF condition THEN sequence
[ELSEIF condition THEN sequence]...
[ELSE sequence] END

```
IF X=0 THEN P:=Q; GOTO STOP END;
IF Y>1 THEN P:=Q ELSE Q:=P END;
IF X<Y THEN
  XX:=YY; J:=K;
ELSEIF X>Y THEN
  XX:=ZZ; J:=L;
END;
```

For, To Statement

FOR identifier := expression [BY expression] TO expression
DO blockbody REP

Increment, limit and initial value evaluated once only in that order as of mode integer; control variable is read only; if BY omitted then increment assumed 1.

```
FOR I:=1 TO 10 DO A(I):=0 REP;
FOR J:=—K BY 2 TO L DO
  CALL(J); A(J):=1;
REP;
```

If control variable not used then use form

```
TO expression DO blockbody REP
```

this repeats body 'expression' times.

```
TO 100 DO OUT('*') REP;
```

While Statement

WHILE condition DO sequence REP

```
WHILE INPUT (3)#1 DO
  DELAY (1500);
  TWRT (''#NL#SWITCH ON PUMP'');
REP;
```

Procedure Statement

Similar to function calls in expressions;

```
TWRT (''FINAL VALUE OF X='');
IWRTF (X,3);
TWRT (''#NL#JOB FINISHES'');
```

Functions may be used for side effects

```
IN ();%DISCARD NEXT CHARACTER%
```

Return Statement

RETURN|RETURN (expression)

The second form is for functions

```
RETURN (IF X=0 THEN 38.7 ELSE Y END);
```

Code Statement

CODE digitlist, digitlist; codeitem . . .

RTL/2 items accessed by prefixing by 'trip 1'; component names and databrick variables followed by 'trip 2' and name of host mode or brick; 'trip 1' and 'trip 2' depend on implementation. Statement terminates with 'trip 1' RTL.

```
CODE 6,0;
  MOV *COUNTER/MISC, *II (5)
* RTL;
```

MODULES

This is the unit of compilation and consists of one or more bricks plus TITLEs, OPTIONs, LET definitions, MODE definitions and external descriptions.
The following example illustrates a complete module.

```
OPTION (1)BC;

TITLE
ILLUSTRATION OF MODULE;

LET NL=10;

EXT PROC (REF ARRAY BYTE) TWRT;

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

MODE PAIR (INT OLD, NEW);

ENT PROC SEARCH (REF ARRAY PAIR P, INT X) INT;
  %SEARCHES ARRAY P FOR OLD ENTRY X AND%
  %RETURNS CORRESPONDING NEW ENTRY%
  %OUTPUTS MESSAGE AND GOES TO ERL IF FAILS%
  FOR I:=1 TO LENGTH P DO
    REF PAIR RP:=P(I);
    IF RP.OLD=X THEN RETURN (RP.NEW) END;
  REP;
  TWRT (''#NL#SEARCH FAILS'');
  GOTO ERL;
ENDPROC;
```

STANDARDS

The following procedures and data bricks should be available to the user in any RTL/2 system. They are shown here as they would appear as external descriptions.

Error Recovery

```
SVC DATA RRERR;
  LABEL ERL;          % unrec error label %
  INT ERN;            % unrec error number %
  PROC (INT) ERP;     % rec error procedure %
ENDDATA;

EXT PROC (INT) RRGEL; % set ERN; monitor; GOTO ERL %
```

Stream I/O

```
SVC DATA RRSIO;
  PROC () BYTE IN; % read next character %
  PROC (BYTE) OUT; % output character %
ENDDATA;

SVC DATA RRSED;
  BYTE TERMCH,      % termination char on input %
  IOFLAG;           % error flag for input %
ENDDATA;

EXT PROC ()FRAC FREAD;      % read fraction %
EXT PROC ()INT IREAD;       % read integer %
EXT PROC ()REAL RREAD;      % read real %

EXT PROC (REF ARRAY BYTE, REF ARRAY BYTE) INT
                                        TREAD;
  % read chars into first array; terminators in second %
EXT PROC (INT) NLS, SPS;    % output newlines, spaces %

EXT PROC (FRAC) FWRT;   % write fraction unformatted %
EXT PROC (INT) IWRT;    % write integer unformatted %
EXT PROC (REAL) RWRT;   % write real unformatted %

EXT PROC (FRAC, INT) FWRTF;   % fraction formatted %
EXT PROC (INT, INT) IWRT;     % integer formatted %
EXT PROC (REAL, INT, INT) RWRTF; % real formatted %

EXT PROC (REF ARRAY BYTE) TWRT; % write text %
```

Formats:

```
FREAD()     [layout] [+|—] digit. digitlist termch
IREAD ()    [layout] [+|—] digitlist termch
RREAD ()    [layout] [+|—] digitlist [.digitlist]
            [E[+|—] digitlist] termch
FWRT (X)    [—] digit.f–digits
IWRT (X)    [—] digits
RWRT (X)    [—] digit.r–digits E[—] digits
```

		field
FWRTF (X, N)	(—| space) digit.N–digits	N+3
IWRTF (X, M)	spaces (—| space) digits	M+1
RWRTF (X, M, 0)	as IWRTF	M+1
RWRTF (X, 0, N)	(—| space) digit.N–digits E (+|—) 2–digits	N+7
RWRTF (X, M, N)	as IWRTF then .N–digits	M+N+2

Error Numbers

unrecoverable	recoverable
1 stack overflow	101 FREAD
2 illegal GOTO	102 IREAD
3 inaccessible ERL	103 RREAD
4 array bound failure	104 TREAD
5 fixed overflow	
6 floating overflow	
7 REAL⟶INT/FRAC overflow	

```
PROC ASK(REF ARRAY BYTE A)INT;
   % OUTPUT MESSAGE A AND RETURN INTEGER REPLY AS RESULT %
   INT N;
AGAIN:
   IOFLAG:=0;
   TWRT(A); OUT(ENQ);
   N:=IREAD();
   IF IOFLAG#0 THEN GOTO AGAIN END;
   RETURN(N);
ENDPROC;

ENT PROC RECIPETASK();
   INT TEMPVAL,I,ITEM;
   ERL:=GIVEUP;                          % UNRECOVERABLE ERROR LABEL    %
TASKSTART:
   WAIT(CTLBEV);

EDITREC:
   I:=ASK("#NL#RECIPE NUMBER");
   IF I<1 OR I>RECIPEMAX THEN GOTO EDITDATA END;
   CURREC:=RECIPE(I);                    % CURREC POINTS TO RECIPE I    %
   ITEM:=ASK("#NL#ITEM");
   TEMPVAL:=ASK(" NEW VALUE");
   IF ITEM=1 THEN
      CURREC.ANR:=TEMPVAL;
   ELSE
      CURREC.CAPBR:=TEMPVAL;
   END;
   GOTO EDITREC;

EDITDATA:
                                         % WRITE OUT COMMON DATA       %
   TWRT("#NL#COMMON DATA IS:#NL#");
   FOR I:=1 TO DATAMAX DO
      IWRT(INT(COMREC(I)*SCALE(I)));     % CONVERT TO CONVERSATION UNITS%
   REP;
   TWRT("#NL#DO YOU WANT TO CHANGE IT#ENQ#");
   IF IN()#'Y' THEN
      TWRT(" BYE");                      % FINISHED                     %
      GOTO TASKSTART;
   END;

   TWRT("#NL# WARNING: ANY CHANGE AFFECTS CURRENT BATCH#NL#");

NEXTITEM:
   I:=ASK("#NL#ITEM NUMBER");
   IF I<1 OR I>DATAMAX THEN GOTO EDITDATA END;
   TEMPVAL:=ASK(" NEW VALUE");
   COMREC(I):=TEMPVAL/SCALE(I);          % CONVERT TO ENGINEERING UNITS %
   GOTO NEXTITEM;

GIVEUP;                                  % ERROR ACTION %
   STOPTASK(THISTASK);
   GOTO TASKSTART;
ENDPROC;
```


Example of RTL/2 application programming

OFFICES & SUBSIDIARY COMPANIES:

Systems Programming GmbH,
6 Frankfurt am Main 18,
Cronstettenstrasse 66,
West Germany,
tel. Frankfurt 557665
telex: 411505

SPL (Italia) SpA.,
via C. Menotti 11,
20129 Milan,
Italy.
tel: 73 86 660

Systems Programming Ltd.,
Svenska A.B.,
Grev Turegatan 35 4 tr.,
114-38 Stockholm,
Sweden
tel: 23 20 93

OY Systems Programming Ltd.,
Lonnrotinkatu 38 A8,
Helsinki 18,
Finland.
tel: 64 37 63

ASSOCIATED COMPANIES:

Steria,
3 rue du Marechal de Lattre de Tassigny,
78-Le Chesnay,
France.

Steriabel,
Rue de Namur,
59/1000 Bruxelles,
Belgium.

Applied Research of Cambridge,
5 Jesus Lane,
Cambridge,
CB5 3BA.
tel: 0223 65015

Systems Programming Pty. Ltd.,
P.O. Box 41165,
Craighall,
Johannesburg,
South Africa.

HEAD OFFICE

12-14 Windmill Street,
London W1P 1HF
tel: 01-636 7833
telex: 21784

MIDLANDS REGION

25 St. James's Street,
Nottingham, NG1 6FH
tel: 0602 45011