I R T B

Industrial Real - Time

BASIC

IRTB

Industrial Real - Time

BASIC

III/1218/81-EN

EWICS TC2 81/8 Sept 1981

IRTB

Industrial Real-time BASIC

Draft Standard

This report describes a draft standard for a Real-time module of BASIC for use in applications such as control, automation and monitoring. The standard takes account of current implementations and practices, and modern trends in language design.

The module was defined by the technical committee on the programming language BASIC of the European Workshop on Industrial Computer Systems (EWICS TC2), in conjunction with the European Computer Manufacturers Association (ECMA TC21) and the American National Standards Institute (ANSI X3J2). It will eventually become part of the ECMA/ANSI BASIC Standard which will be submitted to the International Standards Organisation (ISO).

The intention of this publication is to elicit comments and criticisms from as wide an audience as possible prior to formal standardisation. Comments should be sent by the 15th of January 1982 to the TC2 chairman or document secretary, from whom further copies of this document may be obtained.

J. Szlanko (TC2 chairman)
KFKI
Central Research Inst. for Physics
POB 49
H-1525 Budapest
Hungary

Tel ++36 1 166540
Telex 224722 KFKI H

A. Lewis (TC2 document secretary)
I & AP 347.2
AERE Harwell
Didcot
Oxon. OX11 ORA
England

Tel ++44 235 24141 Ext. 4220
Telex 83135 ATOMHAR

The development of this draft was supported by the Commission of the European Communities, Directorate General III. The views expressed herein are not, however, necessarily those of the Commission.

This document was prepared by the following members of TC2:

G. Bull	Hatfield Poly. Herts. AL10 9AB	UK
M. Bellardinelli	Olivetti & C. S.p.A. I-10015 Ivrea	I
M. Dearlove	Kent Process Control Ltd. Herts. SG4 OTG	UK
G. Ehret	KFK-IAK Postfach 3640, D-7500 Karlsruhe 1	D
A. Jolley	Ferranti Ltd. Manchester M22 5LA	UK
W. Koblitz	Techn. Universitaet A-1040 Wien	A
J.P. Lamoitier	(Consultant) Ave A. Dumas, F-78370 Plaisir	F
A. Lewis	AERE Harwell Oxon OX11 ORA	UK
R. Newton	Teeside Poly. Middlesbrugh Cleveland	UK
J. Szlanko	KFKI POB 49 H-1525 Budapest	H
W. Puczylowski	Inst. of Mathematical Machines, Wasaw	PL
G. Trainito	LADSEB - CNR, I-35100 Padova	I
G. Windal	I.R.I.S. F-59651 Villeneuve-d'Ascq	F
H. Woda	Unicomp GmbH D-7500 Karlsruhe	D

CONTENTS PAGE

1. Introduction

A standard for BASIC is being defined jointly by the American National Standards Institute (ANSI), the European Computer Manufacturers Association (ECMA), and the European Workshop on Industrial Computer Systems (EWICS). The standard will define the core language BASIC together with a number of enhancement modules, one of which is Real-time. The core will include the existing standard for Minimal BASIC (1,2,3) as a subset. Industrial Real-time BASIC consists of the core plus the real-time module and possibly other enhancement modules. The 'Draft Standard' referred to in this document is the ANSI draft for the new Standard.

This document describes Insustrial Real-time BASIC (IRTB). Section 2 describes the main features of the core, and section 5 defines the syntax and semantics of the real-time module in a formal way using the conventions of the ANSI/ECMA Draft Standard.

Some features in the formal definition are specified as 'implementation-defined' (see Appendix 3). An example concerns the details of plant interface equipment accessed in process input and output statements. Process input and output is defined rigorously from the point of view of the application program, but the method of accessing the hardware depends on the equipment used. The documentation for an implementation should define all sections specified as 'implementation-defined' in the standard.

The Draft Standard does not address the problem of building a distributed system. However, careful attention was paid to the design to ensure that a compatible extension could be made to accommodate systems incorporating functional distribution. Appendix 1 in this document describes a set of declarations that will enable a real-time program to run in a distributed system.

2. Main Features of Standard BASIC

Two simple data types are provided - numeric and string, together with one and two dimensional arrays of these data types. Structures can be declared, which are collections of the data types numeric and string, simple values and arrays, in any combination.

Identifiers may be up to 31 characters long (upper and lower case letters, digits and underline). String identifiers are distinguished by having a dollar sign ($) at the end.

The numeric data type is defined to be floating decimal (like a calculator). Powerful string handling is provided together with operations on matrices, comprehensive file input/output, exception handling and debugging facilities.

Selection is provided through the if-then-else and case statements. These take the following form:

```
100 IF condition THEN        100 SELECT expression
110     statement            110 CASE constant
120     statement            120     statement
130     -                    130     statement
140     -                    140     -
150 ELSE                     150 CASE relational-operator constant
160     statement            160     statement
170     statement            170     -
180     -                    180 CASE constant TO constant
190 END IF                   190     statement
                             200     -
                             210 CASE ELSE
                             220     statement
                             230     -
                             240 END SELECT
```

For compatibility with Minimal BASIC selection is also provided by IF condition THEN line-number and other statements that reference line numbers. It is for this reason and for editing purposes that line numbers are required as part of a BASIC program.

Repetition is provided by two constructs - the for-block for definite repetition, and the do-block for indefinite repetition. These take the form:

```
100 FOR i = a TO b STEP c    100 DO WHILE condition
110     statement            110     statement
120     -                    120     -
130 NEXT i                   130 LOOP

200 DO                       200 DO
210     statement            210     statement
220     statement            220     statement
230     -                    230     -
240 EXIT IF condition        240 LOOP UNTIL condition
250     statement
260     -
270 LOOP
```

Three kinds of procedures are provided - subprograms, external functions and internal functions. In addition the Graphics module introduces picture subprograms. Subprograms and external functions communicate with the calling program unit only through the parameter list and the returned value of the function (ie. variables are local to a program unit); internal functions share the same variable space as the surrounding program unit in addition to having parameters. Subprograms and external functions are defined at the end of the program; internal functions are defined within a program unit. Subprograms and functions are defined and called as follows:

```
100 SUB name (formal params)  100 DEF name (formal params)
110     statement             110     statement
120     statement             120     statement
130     -                     130     -
140 END SUB                   140 END DEF

400 CALL name (actual params)   400 LET X = name(actual params)
```

The position of a function definition determines whether it is internal or external.

A lower level of structuring is provided by the GOSUB and RETURN statements.

Comments are introduced through the REM statement or end of line comments which start with an exclamation mark (!).

A Real-time BASIC program consists of a real-time declaration section, a set of parallel activities, and a number of external procedure units.

3. Functional Capability and Rationale for IRTB

A Real-time BASIC program is divided into a number of concurrent single-thread activities which cooperate to achieve the overall objective of the application.

Statements are provided to start concurrent activities, and to enable them to respond to internally or externally generated events. Once started, concurrent activities execute in parallel (at least conceptually).

Each activity is a program module that communicates with its environment through three types of 'ports':

a. process I/O ports that communicate with plant interface hardware,

b. message ports for synchronisation and communication between concurrent activities, and

c. shared-data ports for access to data structures outside the individual activities, for example data in a real-time database system.

The executable code for an activity is written in BASIC. Activities have the usual facilities to access system resources such as files, the computer console and subprograms.

3.1 Concurrent Activities

IRTB is intended for real-time applications that can be described in terms of a number of concurrent activities which are largely independent and asynchronous, but which can communicate and synchronise. The program for such an application does not have an overall thread of control. The program must be capable of running indefinitely - it is not a problem-solving program that starts, operates on some data to produce some output, and is then finished.

A typical application program could be as follows: A number of input activities collect data from external hardware, check the values against limit conditions and store some of the values in shared data. Other activities read the shared data, perform statistical analysis and data reduction and store the results in another section of shared data. Further activities read these results, produce data-logs on demand and archive a summary of the data. This is essentially a problem in concurrent programming since the data is 'pipelined' through the system - archiving activities work on one set of data while the statistical analysis activities are processing the next set, concurrently with the input activities collecting new data and monitoring continuously the state of the plant.

The language requirements are different from those in other parallel-processing environments in which certain aspects of a problem can be processed in parallel, whilst other parts are strictly sequential. In this case 'fork and join' type constructs are appropriate.

The environment is also different from time-shared or multi-user systems where the main requirement is minimum interaction between tasks. In a multi-user system any concurrency should be invisible and is not the concern of an individual user, whereas in real-time systems control of concurrent activities is often the essence of the problem.

The concept of 'Communicating Sequential Processes' (4) is appropriate for 'pipelining' when each parallel section must execute once each time a set of data is available and a set of conditions is true. However, in control and automation applications the activities are more independent. Most of the activities run continuously, occasionally synchronising and communicating with other activities.

It is inappropriate to implement concurrent activities by existing constructs such as subprograms or functions because concurrent activities must be able to call subprograms or functions in the usual way, and the semantics are incompatible. Subprograms are typically called with parameters and return to the calling program at a defined end-point, whereas concurrent activities typically execute in an indefinite loop and have nowhere to return to since they are not called.

In order to define concurrent activities a new language structure for BASIC, the 'parallel section', has been introduced. A parallel section is a program unit in which all variables, internal functions, channel numbers, data-statements etc. are local to the section. Execution of a parallel section constitutes a concurrent activity.

3.2 Data Structures

The concept of a data structure has been introduced to define the interface presented by the three types of ports. A data structure is similar to a record in Pascal for example, in that it is an ordered list of the data types numeric or string, scalar or array. A data structure is an abstract structure in the sense that it does not define data storage and is not associated with particular variables or shared data sections - it is a 'template' that defines the structure of data transferred through a port.

The use of data structures allows a language processor to check the consistency of statements transmitting data through message, shared data and process-I/O ports. It also allows the checking of compatibility between interfaces of communicating activities, particularly when they are in separately compiled program units. For large systems, and especially in the distributed case, the declarations for shared data, message paths and process-I/O paths will be in a separate global section that becomes the 'system definition'. The concept of a data structure will facilitate consistency checking by the language processor between the global section and the code for the individual activities.

3.3 Process Input and Output

The keywords IN FROM and OUT TO are used for statements that perform I/O to plant interface equipment. New keywords are used to distinguish process I/O from conventional I/O. It is important to make the distinction apparent in the program text because process I/O is semantically and functionally different from conventional I/O in the following respects:

a. Process I/O always refers to a unique, identifiable piece of hardware in the process interface system, such as a temperature sensor or a stepping-motor controller. In conventional I/O the nature of the source or destination and the organisation of its data are not relevant to the application program. In other words process I/O is device specific while conventional I/O is device independent.

b. Process declarations are used to establish a static connection between a named process port and a specific piece of hardware. Conventional I/O requires executable open and close statements to establish a temporary association between a channel and an unknown data source or destination.

Further, a system can include a large number of process peripherals, so the identification of process ports by channel number would be no more acceptable than the identification of numeric or string variables by a reference number.

In order to remove the implementation dependent part of an application from the coding of the activities, process I/O statements refer to process port names. Separate declaration statements are used to specify the characteristics of a named port, the method of access to the device connected to it, and the format of its data. The parameters needed to define the access and data format depend on the type of hardware used, so this part of the declaration is one of the areas left as 'implementation-defined'.

Declarations are provided to define arrays of process ports. Sets of logically related process peripherals can be grouped into arrays, for example to allow many input or output operations to be specified in a FOR - NEXT loop. The requirement for process port arrays is similar to the requirement for numeric and string arrays.

3.4 Messages

A message mechanism is provided for synchronising concurrent activities, and for passing data at the point of synchronisation. Message communication is a subset of the Ada (5) 'rendez-vous' mechanism.

Normally two activities participate in a message transfer, the message path being the logical connection between a 'send' port in one activity and a 'receive' port in the other. When both activities reach corresponding send-statements and receive-statements, data are moved from the sending activity to simple variables and/or arrays in the receiving activity. The transmission of the data is an indivisible operation.

A single receive port in one activity can be connected to many send ports in other activities. Because of the synchronising constraints and the indivisibility of message data transfer, this configuration can be used to implement a 'Monitor' (6) for resource management. The sending activities will be forced to queue, the data being accepted from each in turn, allowing that queued activity to proceed. An example is a logging printer activity that accepts data-log information from a number of other activities, with the requirement that the printing of the data from each activity must be completed without interruption before the next set of data is accepted.

Broadcasting of messages from one send port to many receive ports is not permitted. Such a configuration would lead to non-deterministic behaviour of the program since it could not be known how many receive ports were supposed to receive the data. If the message were received only by those activities that had reached receive statements when the send statement is executed, timing variations could cause some activities to miss the information.

3.5 Shared Data

Get-statements and put-statements are used to access data that exists independently of the executing activities. The view of the shared data from the point of view of an activity is declared in data-port declarations. A data-port declaration defines the name of a data port and the structure of the data accessible through it.

The nature of the physical data itself, and how it is stored and managed is not defined in BASIC. The purpose of shared data ports is to provide a mechanism for accessing data whose scope is wider than that of an individual activity. In the simplest case the shared data could be just some locations in common memory. Alternatively, according to the requirements of the application, the data could be part of a database, with the visibility from the shared data ports in the activities controlled by some external mapping, such as a database management system.

Some typical requirements from current applications of IRTB are:

a. Generate periodic backups of the shared data to safeguard the system in the event of a crash.

b. Generate backups at specific points in the application program to provide known recovery points.

c. Optionally use a 'clean' database or use pre-loaded or previous data on system startup.

d. Provide a hierarchical, distributed database management system with different compromises between security and speed of access according to the requirements of different sections.

e. Use the database as the interface to other, non-BASIC, parts of the system such as autonomous analogue scanning sub-systems or a higher level artificial intelligence control program written in Pascal.

3.6 Events

Interrupt servicing, with all the attendant problems of saving and restoring context, is not provided in IRTB. Hardware attention signals, which generate program interrupts at levels of software below BASIC, become 'events' that can be 'waited for' by concurrent activities.

The service routine for an event is an activity with a wait-statement naming the event. After servicing the event, the routine returns to the wait-statement to await the next occurrence of the event. In this way the concurrent activity is effectively the interrupt service routine, but it is scheduled like any other activity and all the details of saving and restoring context are handled by the system.

An event can also be set by the software using a signal statement. This facility provides an alternative method of synchronising concurrent activities. A significant difference between signal-statements and send-statements is that a signalling activity continues and does not wait for the receiving activity to act upon the event, whereas an activity executing a send-statement waits until the receiving activity accepts the data.

The signal-statement is also useful for testing application software without using the external hardware.

No 'clear event' statement is provided. An event is cleared automatically when it has caused an activity to proceed from a wait-statement. It follows that there is a one-to-one correspondence between the setting of an event by the hardware or a signal-statement, and a wait-statement that 'consumes' the event. This definition of events provides a facility that encourages the writing of secure, deterministic programs that are easy to understand.

Binary or multi-valued semaphores for example have not been provided because these would need different statements from those defined for handling hardware generated events, and the statements provided, together with the message mechanism, are sufficient for the synchronisation requirements.

3.7 Exception Handling

A large number of exception conditions are defined by the Draft Standard. Most exceptions are fatal and the default action in the absence of a user-written exception handler is to report the exception and stop execution of the concurrent activity in which it occurs. A few exceptions are non fatal; for each of these the Draft Standard defines a specific recovery action. An example of a non fatal exception is providing a non-valid numeric string as the numeric input-reply to an input statement.

The main purpose of exception handling in a real-time BASIC program is to provide the possibility of recovering from an exception condition in a user-written handler, and then continuing program execution. This feature is important for the type of control and monitoring application envisaged for IRTB, in which the program runs indefinitely and must be resilient to hardware failures and exceptions.

Many exception handlers can exist, but only one can be enabled in each program unit (ie. in each parallel activity and each external procedure). If an exception handler is enabled then all exceptions, fatal and non fatal, cause a branch to the first line of the handler. Within the handler, two functions are available to determine the cause of the exception: EXTYPE that returns the exception code number (see Appendix 2) and EXLINE that returns the line number of the statement causing the exception.

There are four ways to leave an exception handler. A CONTINUE statement returns to the statement following the one that caused the exception, and is used when recovery action has been taken in the handler (eg. default values have been supplied after a RECEIVE statement has timed out). A RETRY statement returns to the beginning of the statement that caused the exception, and is used when the condition causing the exception has been corrected in the handler (eg. by making the argument of a square-root function call positive or correcting a file-name string for an OPEN statement). If the END HANDLER statement is reached, then the default system action is invoked. Finally, a RESUME statement is provided that can return to a line-number specified in the enable-handler statement.

Exception handling in BASIC differs from that in Ada where an exception causes a branch to code at the end of the current block, and thence to the context of the surrounding outer block. It is not possible to return directly to the code within the block that caused the exception. This approach is not appropriate for IRTB because a parallel section is not contained within an outer block - it is an independent program module that must continue to run normally after successful recovery from an exception condition.

3.8 Distributed Systems and Independent Compilation

In this document the term 'distributed systems' means application configurations comprising multiple processors without shared memory. If a program is written and compiled as a single unit, the distributed system requires no change to the language except for declarations to specify where the activities are to be executed.

If the program is segmented into independently compiled sections, then there must be a global section containing declarations for message paths connecting message ports in the separately compiled sections. It is convenient if the global section also contains the configuration description specifying the distribution of activities among the processors, and the description of the global shared data.

Note that the requirement for a global section comes from the need for independent compilation, regardless of whether the activities run in a distributed or a non-distributed configuration. The global section does not contain executable code, it comprises a set of static declarations that are effectively a 'system description' describing the intercommunication between the activities.

Appendix 1 gives more details of the extension to distributed applications and independent compilation.

4. The Language Definition

4.1 Conventions

The conventions used in the formal definitions in section 5 are those employed in the relevant ECMA and ANSI standards. The conventions are explained fully in those documents, but a brief description of the method of syntax definition is given below.

The syntactic metalanguage used to define the syntax of IRTB is derived from Backus-Naur Form (BNF). The IRTB syntax is defined by a series of 'production rules' that define syntactic elements of the language in terms of other syntactic elements in a hierarchical manner, until a 'terminal symbol' is reached. A terminal symbol is typically a single character of the language being defined, ie. IRTB. Certain special symbols are used whose meaning is defined below:

The symbol = is interpreted as meaning 'is defined as' if only one definition is given, or 'is defined as either' if there is more than one definition. In the latter case the symbol / is interpreted as meaning 'or'.

> is like '=' above, but it is used when the production rule augments another production. It can be read as 'includes'.

? the preceding syntactic element is optionally present.

* the preceding syntactic element is optionally present an arbitrary number of times (including zero times).

(and) are used to group syntactic elements into a single unit.

/ separates alternatives.

Spaces and new lines are used to improve legibility of the definitions; they have no syntactic significance.

The following example illustrates the use of some of these symbols:

```
out-structure           = out-structure-element
                          (comma out-structure-element)*
out-structure-element   = expression / formal-array
```

This means that an out-structure is a list of out-structure-elements. If there is more than one item in the list, the items are separated by commas. Each item can be either an expression or a formal array. An example of an out structure satisfying this definition is:

A + 2, B(), C$

The words 'may' and 'shall' have precise meanings in the formal definitions. The word 'may' is used in a permissive sense to indicate that a standard-conforming implementation may or may not provide a particular feature. The word 'shall' is used in an imperative sense to indicate that a program is required to be constructed, or that an implementation is required to act as specified in order to meet the constraints of standard conformance.

4.2 Assumed definitions

The formal definitions in Section 5 concern only the Real-time module. It is assumed that it is an extension of BASIC as defined in the ECMA/ANSI Draft Standard or at least that it uses a 'host' with similar facilities. The following definitions are referred to directly or indirectly in section 5 and are some examples from a typical BASIC host language definition.

```
line              = line-number statement tail
line-number       = digit digit? digit? digit?
digit             = 0 / 1 / 2 / 3 / 4 / 5 / 6 / 7 / 8 / 9
statement         = data-statement / def-statement /
                    dimension-statement / gosub-statement /
                    goto-statement / if-then-statement /
                    input-statement / let-statement /
                    on-goto-statement / print-statement /
                    randomise-statement / read-statement /
                    remark-statement / restore-statement /
                    return-statement / stop-statement
tail              = tail-comment? end-of-line
tail-comment      = ! remark-string
end-of-line       = implementation-defined
remark-line       = line-number
                    (null-statement / remark-statement)
                    end-of-line
remark-statement  = REM remark-string
subscript-part    = index (comma index)?
index             = numeric-expression

block             = (line / for-block)*
for-block         = for-line for-body
for-body          = block next-line
for-line          = line-number for-statement tail
next-line         = line-number next-statement tail
for-statement     = FOR control-variable equals-sign
                    initial-value TO limit
                    (STEP increment)?
control-variable  = simple-numeric-variable
initial-value     = numeric-expression
limit             = numeric-expression
increment         = numeric-expression
next-statement    = NEXT control-variable
procedure-part    = remark-line* procedure
```

```
numeric-rep      = significand exrad
significand      = integer full-stop? / integer? fraction
integer          = digit digit*
fraction         = full-stop integer
exrad            = E sign? integer
sign             = + / -
```

A real-time-program is a sequence of lines. Each line contains a unique line-number which facilitates program editing and serves as a label for the statement contained in that line.

The values of the integers represented by the line numbers shall be positive and non-zero, leading zeroes shall have no effect. Lines shall occur in ascending line number order.

It is assumed that the function TIME$ defined in the Draft Standard is available. This function returns a string of the form "hrs:mins:secs" where hrs, mins and secs are each 2 characters long. The range of values for hrs is "00" to "23" and for mins and secs is "00" to "59". An example of a value for TIME$ is "17:59:01".

4.3 Conformance

The Draft Standard gives a set of conformance rules for programs and implementations. The rules are intended to ensure that a program conforming to the program conformance rules will produce the same results on any implementation conforming to the implementation conformance rules. In the case of IRTB this ideal may not be realisable because it is not possible to define the real-time performance of an implementation and because a real-time-program does not usually produce 'results' in the sense of a data processing program. However, programs written in IRTB and implementations of IRTB should follow the conformance rules with respect to section 5 of this document. The conformance rules are as follows.

A program conforms to the Standard only when

- the program and each statement or other syntactic element contained therein is syntactically valid according to the syntactic rules specified by the Standard, and

- the program as a whole violates none of the global constraints imposed by the Standard on the application of the syntactic rules.

An implementation conforms to the standard only when

- it accepts and processes all programs conforming to the Standard,

- it reports reasons for rejecting any program that does not conform to the Standard,

- it interprets errors and exceptional circumstances according to the specifications of the Standard,

- it interprets the semantics of each statement of a conforming program according to the specifications in the Standard,

- it interprets the semantics of a conforming program as a whole according to the specifications in the Standard,

- it accepts as input, manipulates and can generate as output numbers of at least the precision and range specified in the Standard,

- it is accompanied by documentation that describes the actions taken in regard to features referred to as "implementation-defined" in the Standard, and

- it is accompanied by documentation that describes and identifies all enhancements to the language defined in the Standard.

5. Formal Definition of the Real-time Module

The real-time module in this document is part of the proposed joint ANSI/ECMA/EWICS Standard for BASIC. The language is intended for use in applications involving control, automation, and monitoring. It enables a program to be divided into a number of concurrent single-thread activities which cooperate to achieve the overall objective of the application.

Facilities are provided to schedule execution of concurrent activities so that they may respond to both internally and externally generated events. Communication between concurrent activities is possible either through the use of shared data or by the transmission of messages.

An activity can communicate with process objects which are a part of the external environment of a real-time-program. Typical process objects are measurement or control points in a plant interface. Communication between a concurrent activity and a process object is accomplished by input and output operations accessing the process object through a process port.

An implementation-defined scheduler shall determine which of those concurrent activities in progress shall actually be executing. Implementations may interrupt the execution of a concurrent activity in order to prevent excessive delays in the execution of other concurrent activities.

Access to files and procedures (external functions, subprograms and pictures) from different concurrent activities is not synchronised by the system. Since procedures may be called from more than one concurrent activity they shall be reentrant.

5.1 Real-time programs

5.1.1 General Description

A real-time-program is composed of real-time declarations (cf. Section 5.2) that describe a process environment, one or more parallel-sections, and some number of procedures which may be invoked by these parallel-sections. Each parallel-section is named and is delimited by the keywords PARACT (PARallel ACTivity) and END PARACT. Parallel-sections constitute separate program-units and serve to define concurrent activities.

Execution of a parallel-section is enabled by a scheduling-statement (cf. Section 5.3) and starts at the first line of the section.

Execution of each statement is completed before execution of the next statement in sequence in the same parallel-section is started, except that a statement may be interrupted by the occurrence of a non-fatal exception which causes a user-defined exception handler to be invoked which does not, however, handle the exception (see section 6).

5.1.2 Syntax

```
1.  real-time-program       = real-time-declarations
                              parallel-section parallel-section*
                              procedure-part*
2.  program-unit            > parallel-section
3.  parallel-section        = remark-line* paract-line
                              block* end-paract-line
4.  line                    > paract-line / end-paract-line
5.  paract-line             = line-number paract-statement tail
6.  paract-statement        = PARACT routine-identifier
                              (URGENCY urgency)?
7.  routine-identifier      = letter identifier-character*
8.  urgency                 = integer
9.  end-paract-line         = line-number end-paract-statement
                              tail
10. end-paract-statement    = END PARACT
11. statement               > real-time-statement
12. real-time-statement     = parstop-statement /
                              scheduling-statement /
                              process-io-statement /
                              data-io-statement /
                              message-io-statement
13. parstop-statement       = PARSTOP
```

A given routine-identifier shall not occur in more than one paract-statement in a real-time-program.

Control-statements shall refer only to lines in the parallel-section in which they occur. Real-time-statements shall occur only in parallel-sections.

5.1.3 Examples

2.
```
320 PARACT RIG1
330     WAIT TIME 17*60*60
340 PRINT "TIME TO GO HOME"
350 END PARACT
```

5.1.4 Semantics

Execution of a parallel-section in a real-time-program shall constitute a concurrent activity. At any point in the execution of a real-time-program, a concurrent activity may be in one of the following states:

- in progress, ie., in the initial state of the concurrent activity defined by the lexically first parallel-section, or in the state of a concurrent activity following execution of a start-statement naming that activity; or

- stopped, ie., not yet in progress, or formerly in progress but subsequently terminated by execution of a parstop-statement, an end-paract-statement, or a statement generating a fatal exception which is not inhibited by the action of an exception handler; or

- waiting, ie., formerly in progress but suspended by execution of a wait-statement or message-io-statement, until the occurrence of a specified event, the passing of a specified length of time, the arrival of a specified time of day or the exchange of messages.

Several concurrent activities may be in progress at any given time. Initially the only concurrent activity in progress shall be that defined by the lexically first parallel-section in the real-time-program; other concurrent activities shall be placed in progress only by the execution of start-statements (cf. Section 5.3).

The urgency of a parallel-section shall indicate to the scheduler the relative importance of the concurrent activity. A lower value shall indicate a greater importance. The precise interpretation of the urgency shall be implementation-defined.

At the initiation of the execution of a parallel-section the values of all variables shall be implementation-defined.

Lines in a parallel-section shall be executed in sequential order, starting at the first line of the parallel-section, until

- some other action is dictated by the execution of a line, or
- an exception occurs, or
- a stop-statement, chain-statement, parstop-statement, or an end-paract-statement is executed.

Execution of a parstop-statement or of an end-paract-statement shall terminate execution of the concurrent activity in which it occurs, causing that activity to stop until placed in progress again by another execution of a start-statement. Execution of a stop-statement or a chain-statement shall terminate execution of the entire real-time-program. The occurrence of a fatal exception that is not handled by an exception-handler shall stop the concurrent activity in which it occurs.

Each parallel section is a distinct entity in that identifiers used to name variables, arrays, internal functions and exception handlers shall be local to the section, ie. they shall name different objects in different parallel sections. Identifiers used to name supplied functions, parallel sections, procedures defined as program units, process I/O ports, process-port-arrays, message ports and shared data ports shall be global to the entire real-time program, ie. they shall name the same object wherever they occur.

5.1.5 Exceptions

None.

5.1.6 Remarks

Execution of a concurrent activity may be interrupted at implementation-defined times in order to execute other concurrent activities which are in progress.

Possible interpretations of the urgency of a parallel-section might be the priority of that section or a deadline for execution of the section.

5.2 Real-Time Declarations

5.2.1 General Description

Concurrent activities communicate with the external environment through process ports. Process port declarations define the names of these ports and the attributes of process-objects in a real-time system attached to these ports. Process-objects may be either active or passive. Passive process-objects are typically measurement and control points in a plant interface, such as temperature sensors or stepping motor controllers (cf. section 5.4). Active process-objects, or process-events, are typically sources of program interrupts, such as timers and alarms (cf. section 5.3).

Data ports provide a means of accessing data whose scope is wider than an individual concurrent activity. A data port declaration defines the name of a data port and the structure of the data accessible through it (cf. Section 5.5).

Message ports provide a means of transferring data between two concurrent activities; the data transferred does not exist outside the two activities. A message port declaration defines the name of a message-port and the structure of the data transferred through it. A message is sent when the same message-port-name is used in two concurrent activities, in a send-statement in one and a receive-statement in the other (cf. Section 5.6).

Data structure declarations provide a means of specifying the structure of data transferred through process, data and message ports. They enable a language processor to check the validity of statements sending and receiving data through a port, and they specify indivisible units of shared data.

5.2.2 Syntax

```
1. real-time-declarations   = (remark-line / declaration-line)*
2. declaration-line         = line-number declaration-statement
                              tail
3. declaration-statement    = data-structure-dec /
                              process-dimension-statement /
                              process-port-dec /
                              data-port-dec / message-port-dec
4. data-structure-dec       = STRUCTURE structure-name colon
                              repeat-count? type
                              (comma repeat-count? type)*
5. structure-name           = letter identifier-character*
6. repeat-count             = integer OF
7. type                     = (NUMERIC / STRING) dimensioning?
8. dimensioning             = left-parenthesis bounds
                              right-parenthesis
9. bounds                   = integer (comma integer)?
```

```
10. process-dimension-statement = PRODIM process-array-dec
                                  (comma process-array-dec)*
11. process-array-dec           = process-port-array dimensioning
12. process-port-array          = letter identifier-character*
13. process-port-dec            = PROCESS
                                  (process-clause / event-clause)
                                  access-information?
14. process-clause              = io-qualifier (process-port-name /
                                  process-port-array dimensioning)
                                  (OF structure-name)?
15. io-qualifier                = INPUT / OUTPUT / OUTIN
16. process-port-name           = letter identifier-character*
17. event-clause                = EVENT event-name
18. event-name                  = letter identifier-character*
19. access-information          = string-constant
20. data-port-dec               = SHARED data-port-name
                                  dimensioning? OF structure-name
21. data-port-name              = letter identifier-character*
22. message-port-dec            = MESSAGE message-port-name
                                  OF structure-name
23. message-port-name           = letter identifier-character*
24. line                        > declaration-line
```

Any structure-name appearing in a process-clause, data-port-dec or message-port-dec shall be defined in a data-structure-dec in a lower-numbered line. The scope of process-port-names, process-port-arrays, data-port-names and message-port-names shall be all the parallel sections in a real-time-program; any such identifier shall be declared in at most one declaration-statement.

The value of the integer in a repeat-count shall be greater than zero.

For each process-port-array, there shall be a process-port-dec for every element of that array. The elements shall all have the same io-qualifier and the same data-structure (if any).

The value(s) of the integer(s) in the dimensioning in a process-array-dec shall be greater than zero. A process-port-array occurring in a process-port-dec must be declared in a process-array-dec in a lower numbered line. The dimensioning in a process-clause shall have the same number of dimensions and take values between one and the value of the corresponding dimension in the process-array-dec.

5.2.3 Examples

```
4.  STRUCTURE OPR: STRING, 2 OF NUMERIC, NUMERIC(10)
    STRUCTURE A1: 2 OF NUMERIC
    STRUCTURE B1: NUMERIC
10. PRODIM RIG1(3), RIG2(3)
13. PROCESS INPUT WEIGHT OF A1 "ADCCHAN 3"
    PROCESS OUTIN PANEL OF OPR "Q, 177640"
    PROCESS INPUT A TIMEOUT 4 "BCD 4"
    PROCESS OUTPUT Z1 OF B1
    PROCESS OUTIN RIG1(2) "U, 166000"
    PROCESS EVENT FULL "INT 36"
19. SHARED FLIGHT(10) OF OPR
    SHARED D OF B1
21. MESSAGE LINK OF OPR
```

5.2.4 Semantics

A data-structure-dec shall declare the name of a data structure for use in process-port-decs, data-port-decs and message-port-decs. A data structure is an abstract structure (ie. one without any storage allocated to it) consisting of an ordered list of types which may be either numeric or string, scalar or array. A repeat-count shall specify the number of occurrences of the type that follows it.

Each process-array-dec in a process-dimension-statement shall declare an array of process-ports. The array shall be one-dimensional or two-dimensional according to whether one or two integers are specified in the bounds. In addition, the bounds specify the maximum values of expressions used as subscripts for the array. The minimum value of an expression used as a subscript for a process port array shall be one.

A process-port-dec shall define the name of a process port and the attributes of a process-object in a real-time system attached to thet port. The bounds following a process-port-array shall be interpreted as a subscript-part, and the process-port-dec shall define the attributes of the process-object attached to that element of the process-port-array.

The presence of a process-clause shall indicate that the process-object attached to that process port is passive. The io-qualifier in the process-clause shall indicate the permitted directions of data transfer through the port: INPUT shall indicate that the process-object provides input only, OUTPUT that it accepts outpu only, and OUTIN that it supports both input and output.

The validity of in-structures and out-structures in process-io-statements shall be checked by the language processor by reference to the structure-name in the corresponding process-clause. In the absence of a structure-name in the process-clause, the default data structure shall be a single numeric.

The presence of an event-clause in a process-port-dec shall declare the named process-object to be active, ie. to be a process-event. When connected, a process-event shall be capable of generating events which return concurrent activities waiting for them to the state of being in progress (cf. Section 5.3).

Access-information for a process port specifies a particular process-object attached to that port and the format of its data. Access information for an active process-object typically specifies the source of a hardware interrupt signalling the occurrence of an event associated with that object together with information about how to control the interrupt. The interpretation of the access information shall be implementation-defined.

A data-port-dec shall define the name of a data port and the structure of the data accessible through it. If a dimensioning appears in a data-port-dec, then it shall define an array of instances of the given structure. The array so defined shall be either one-dimensional or two-dimensional according to whether one or two integers are specified in the bounds. If no dimensioning appears, a single instance of the given structure shall be defined. Shared data shall be accessible by all concurrent activities (cf. Section 5.5).

A message-port-dec shall define the name of a message port and the structure of the data transferred through it.

5.2.5 Exceptions

None.

5.2.6 Remarks

Process-port-arrays can only be arrays of passive process-objects, ie. arrays of process-events are not permitted.

The format information in the access-information for a process-port may allow the implementation to perform automatic data transformation, such as scaling or conversion between BCD in a process-object and a floating-point internal representation. An implementation may also allow names of routines in the access-information so that special devices can be handled by standard mechanisms invoked automatically each time a process-port is accessed. These routines could, for example, handle access via a multiplexer with a long switching time or handle special Gray code devices.

5.3 Scheduling

5.3.1 General Description

The scheduling requirements for concurrent activities are specified by execution of start-statements and wait-statements. A start-statement places a concurrent activity in progress. The actual execution of concurrent activities which are in progress is scheduled by the implementation according to the urgency of these activities. A wait-statement can be used to suspend execution of a concurrent activity for a specified period of time, until a given time, or until a specified event occurs. Events may be generated externally by connected process-objects or internally by execution of signal-statements.

Connect-statements and disconnect-statements referring to events are used to enable and disable specific event signals from the external hardware.

5.3.2 Syntax

```
1. scheduling-statement  = start-statement / wait-statement /
                           signal-statement / connect-statement /
                           disconnect-statement
2. start-statement       = START routine-identifier
3. wait-statement        = WAIT (wait-time / wait-interval /
                           wait-event)
4. wait-time             = TIME time-expression
5. time-expression       = numeric-time-expression /
                           string-time-expression
```

```
 6. numeric-time-expression  = numeric-expression
 7. string-time-expression   = string-expression
 8. wait-interval            = DELAY numeric-time-expression
 9. wait-event               = EVENT event-name timeout-expression?
10. timeout-expression       = TIMEOUT numeric-time-expression
11. signal-statement         = SIGNAL event-name
12. connect-statement        = CONNECT EVENT event-list
13. event-list               = event-name (comma event-name)*
14. disconnect-statement     = DISCONNECT EVENT event-list
```

An event-name that does not occur in a process-port-dec shall not occur in a connect-statement or a disconnect-statement.

An event-name that occurs in a process-port-dec shall not occur in a signal-statement.

A routine-identifier that occurs in a start-statement shall also occur in some paract-line in the program. An event-name that occurs in a wait-statement shall occur in a signal-statement or shall be declared as a process-event in a process-port-dec.

5.3.3 Examples

```
 2. START FILL
 3. WAIT DELAY 1.5*60*60
    WAIT TIME "09:15:00"
    WAIT EVENT READY TIMEOUT 4
    WAIT TIME A$
12. SIGNAL READY
13. CONNECT EVENT FULL
15. DISCONNECT EVENT FULL, TOOFUL
```

5.3.4 Semantics

Execution of a start-statement shall place in progress the concurrent activity defined by the named parallel-section. Execution of a wait-statement shall cause the concurrent activity in which it occurs to be suspended for a specified period of time, until a specified time, or until a specified event occurs.

The value of a numeric-time-expression shall be interpreted as specifying a number of seconds. If the value of the expression is not an integer, then the accuracy of the time expression is dependent on the resolution of the timer. The value of a string-time-expression shall conform to the format, range of values and interpretation of the TIME$ function (cf. section 4.2).

If a wait-statement specifies a wait-interval, then the concurrent activity shall be suspended for the specified length of time, being placed in progress again when that time has elapsed. If a wait-statement specifies a wait-time with a numeric time-expression, then the concurrent activity shall be suspended until the specified number of seconds have elapsed since the previous midnight, at which time it shall be placed in progress again. If the number of seconds since the previous midnight have already elapsed, then the concurrent activity shall wait until that time the following day. If a wait-statement specifies a wait-time with a string-time-expression, then the concurrent activity shall be suspended until the specified time of day, at which time it shall be placed in progress again. If the specified time of day has already passed then the concurrent activity shall wait until that time the following day.

If a wait-statement specifies a wait-event, then the concurrent activity shall be suspended until that event occurs, at which time it shall be placed in progress again (cf. sections 5.2 and 5.4). If a timeout expression is specified in a wait-event, then an exception shall occur if the specified event has not occurred within the specified length of time.

Execution of a signal-statement shall cause the specified event to occur. Following execution of a signal-statement the concurrent activity continues to be in progress.

Execution of a connect-statement shall cause the specified process-event to be connected. A connected process object can cause events to occur.

Execution of a disconnect-statement shall cause the specified process-event to be not connected, and shall cause any previous occurrence of the event not acted upon by a wait-statement to have not occurred. A process object that is not connected cannot cause events to occur.

An event that has occurred shall place in progress again a concurrent activity waiting for the event. If no concurrent activity is waiting for the event, then the first concurrent activity subsequently to execute a wait-statement naming that event shall remain in progress. In either case, the event shall then be deemed to have not occurred.

If more than one concurrent activity is waiting for the same event, then which one of those activities that shall be placed in progress upon occurrence of that event shall be determined by the underlying system. Only one concurrent activity shall be placed in progress upon each occurrence of an event.

If a new event is caused by a signal-statement before a previous occurrence of the same event has been acted upon by a wait-statement, then that signal-statement shall cause an exception. The events shall then be deemed to have not occurred.

If a new event is generated by a connected process-object before a previous event generated by that object has been acted upon by a wait-statement, then the next wait-statement to be executed that names that event shall cause an exception. The events shall then be deemed to have not occurred.

At the initiation of execution of a real-time-program, all events shall have not occurred, and all process-events shall be not connected.

5.3.5 Exceptions

A start-statement is executed that specifies a concurrent activity that is not stopped (fatal).

A signal-statement is executed that specifies an event that has already occurred, but which has not yet caused a waiting concurrent activity to be placed in progress again (fatal).

The value of a numeric-expression used as a time-expression exceeds 86400, the number of seconds in a day, or is less than zero (fatal).

The value of a string-expression used as a time-expression does not conform to the format of the TIME$ function (fatal).

The event specified in a wait-statement does not occur within the period of time specified in a timeout-clause (fatal).

A new event is generated by a connected process-object before a previous event generated by the object has resulted in a waiting concurrent activity being placed in progress again (fatal).

5.3.6 Remarks

When the system clock requires adjustment, such as for seasonal time changes or to correct for errors, problems can arise with wait-statements specifying wait-times. In particular, if the clock is moved back, any activities that were released from a wait-time during the previous occurrence of that time should not be put in progress again until the following day. Similarly, if the clock is advanced, activities waiting for a time that is 'passed over' should be put in progress as if that time had occurred.

5.4 Process Input and Output

5.4.1 General Description

In-statements and out-statements are used to move data over communication paths between passive process-objects and a real-time-program. An in-statement permits external values to be transferred to program variables, and an out-statement permits the transfer of values to external process-objects.

5.4.2 Syntax

```
1. process-io-statement   = in-statement / out-statement
2. in-statement           = IN FROM (process-port-name /
                            process-port-array subscript-part)
                            TO in-structure timeout-expression?
3. in-structure           = in-structure-element
                            (comma in-structure-element)*
4. in-structure-element   = variable / formal-array
5. out-statement          = OUT TO (process-port-name/
                            process-port-array subscript-part)
                            FROM out-structure timeout-expresson?
6. out-structure          = out-structure-element
                            (comma out-structure-element)*
7. out-structure-element  = expression / formal-array
```

Any process-port-name or process-port-array occurring in an in-statement or out-statement shall be declared in a process-port-dec.

The number and types of elements within an in-structure or out-structure shall conform to the data-structure-dec for the structure specified in the declaration for the corresponding process port, or to the default if no structure-name occurred in the process-port-dec.

5.4.3 Examples

```
2.  IN FROM WEIGHT TO X, Y
    IN FROM PANEL TO A$, B, C, F()
    IN FROM RIG1(NEXT) TO ALPHA  TIMEOUT 2.5
5.  OUT TO Z1 FROM B*C+X
    OUT TO PANEL FROM A$&B$, JIM, FRED, C()
```

5.4.4 Semantics

Execution of an in-statement shall cause values to be obtained from the specified process-port and to be assigned to the corresponding variables and arrays in the in-structure. No assignment of values from the process-object shall take place until the values supplied have been validated with respect to the allowable range for each value and the number of values. If a numeric value causes an underflow, then its value shall be replaced by zero. Subscripts in an in-structure shall be evaluated after values have been assigned to the variables and arrays preceding them(ie. to the left of them) in the in-structure.

Execution of an in-statement shall be regarded as complete only when all values have been assigned to the variables and arrays in the in-structure or when a fatal exception occurs, such as one caused by incorrect data or a hardware failure, or the number of seconds specified by the timeout-expression has expired.

Execution of an out-statement shall cause the expressions in the out-structure to be evaluated and their values, together with the values of all elements in the specified formal-arrays, to be transmitted to the specified process-port.

Execution of an out-statement shall be regarded as complete only when all values from the out-structure have been validated and accepted by the process environment or when a fatal exception occurs, such as one caused by incorrect data or a hardware failure, or the number of seconds specified by the timeout-expression has expired.

The occurrence of a formal-array in an in-structure or an out-structure shall cause the contents of the entire array with that name to be input or output.

5.4.5 Exceptions

The assignment of a value to a numeric-variable or numeric-array in an in-structure causes a numeric overflow (fatal).

The assignment of a value to a string-variable or string-array in an in-structure causes a string overflow (fatal).

The current sizes of the dimensions of a formal-array used in an in-structure or an out-structure do not conform to the data-structure-dec for the structure specified in the declaration for the indicated process-port (fatal).

Execution of an in-statement or an out-statement has not been completed before the timeout given by the timeout-expression has expired (fatal).

A subscript for a process port is not within the range specified by the process-array-dec (fatal).

5.4.6 Remarks

Implementation-defined exception conditions may exist. These are mainly concerned with the characteristics of particular process-objects.

Validation of data obtained from process-objects as required by section 5.4.4 may be subject to implementation-defined limitations. For example, corruption of a string datum may be inherently undetectable.

5.5 Shared Data

5.5.1 General Description

Get-statements and put-statements are used to transmit data between concurrent activities and collections of shared data. The data are transmitted through data ports.

5.5.2 Syntax

```
1.  data-io-statement    = put-statement / get-statement
2.  put-statement        = PUT TO data-port-name subscript-part?
                           FROM out-structure
3.  get-statement        = GET FROM data-port-name subscript-part?
                           TO in-structure
```

Any data-port-name occurring in a put-statement or get-statement shall be declared in a data-port-dec. A subscript-part shall follow the data-port-name if and only if a dimensioning occurs in the data-port-dec for that data-port-name; in that case, the number of subscripts in the subscript-part shall equal the number of dimensions specified by the dimensioning. The number and types of elements within an in-structure or out-structure shall conform to the data-structure-dec for the structure specified in the data-port-dec for the data-port-name.

5.5.3 Examples

2. PUT TO FLIGHT(N+1) FROM I$, N, 2, P()
3. GET FROM D TO E

5.5.4 Semantics

Execution of a put-statement shall cause the expressions in the out-structure to be evaluated and their values, together with the values of all elements in the specified formal-arrays, to be transmitted to the appropriate collection of the shared data.

Execution of a get-statement shall cause the variables and arrays in the in-structure to be assigned values from the appropriate collection of shared data. No assignment of values shall take place until all values have been validated with respect to the allowable range of each value, and the number of values. Subscripts in an in-structure shall be evaluated after values have been assigned to the variables and arrays preceding them (ie. to the left of them) in the in-structure.

Execution of a put-statement or a get-statement shall be regarded as complete when all values have been verified and transmitted, or when a fatal exception has occurred. No other concurrent activity shall access the specified collection of shared data until execution of a get-statement or put-statement is complete.

5.5.5 Exceptions

The assignment of a value to a numeric-variable or numeric-array in an in-structure causes a numeric overflow (fatal).

The assignment of a value to a string-variable or string-array in an in-structure causes a string overflow (fatal).

The current sizes of the dimensions of a formal-array used in an in-structure or an out-structure do not conform to the data-structure-dec for the structure specified in the declaration for the indicated process-port (fatal).

A subscript for a data-port is not within the range specified by the data-port-dec (fatal).

5.5.6 Remarks

None.

5.6 Message Passing

5.6.1 General Description

Send-statements and receive-statements are used to transmit data between concurrent activities. The data are conveyed over message paths which connect a message output port in a send-statement in one concurrent activity to a message input port in a receive-statement in another.

A message path is established at run-time implicitly by the use of the same message port name in two concurrent activities, in a send-statement in one and in a receive-statement in the other.

5.6.2 Syntax

```
1. message-io-statement   = send-statement / receive-statement
2. send-statement         = SEND TO message-port-name
                            FROM out-structure timeout-expression?
3. receive-statement      = RECEIVE FROM message-port-name
                            TO in-structure timeout-expression?
```

The number and types of elements in the out-structure in a send-statement shall match the number and types of elements in the in-structure in any receive-statement specifying the same message-port-name; in addition, they shall conform to the data-structure-dec for the structure specified in the messsage-port-dec for that message-port-name.

A parallel-section shall not contain both a send-statement and a receive-statement specifying the same message-port-name.

5.6.3 Examples

```
2. SEND TO LINK FROM "FIRST", X/2, 17.35, RESULTS()
3. RECEIVE FROM LINK TO A$, P(1), P(2), I() TIMEOUT 30
```

5.6.4 Semantics

A message port in one concurrent activity shall be connected to a message port in another concurrent activity by the execution of a send-statement or a receive-statement in the one concurrent activity using the given message-port-name and the subsequent execution in the other concurrent activity of a receive-statement or a send-statement using the same message-port-name.

Execution of a send-statement or a receive-statement shall not be complete until the specified message port has been connected as a result of executing a corresponding receive-statement or send-statement in another concurrent activity, or an exception occurs.

When such a connection has been made, the expressions in the out-structure in the send-statement shall be evaluated, and their values, together with the values of all arrays in the out-structure, shall be assigned to the variables and arrays in the in-structure in the corresponding receive-statement.

Subscripts in an in-structure shall be evaluated after values have been assigned to the variables and arrays preceding them (ie. to the left of them) in the in-structure.

If a timeout is specified in a send-statement or a receive-statement, then an exception shall occur if no connection is made within the specified length of time.

If a send-statement times out then its message is no longer available for a receive-statement.

If a send-statement is executed and more than one other concurrent-activity is waiting to receive a message through a message port with the same name, then which one of those activities that receives the message shall be determined by the underlying system.

5.6.5 Exceptions

The current sizes of the dimensions of an array used in an in-structure in a receive-statement do not match those of the corresponding array in the out-structure in a send-statement (fatal).

Execution of a send-statement or receive-statement has not been completed before the time specified in a timeout has expired (fatal).

5.6.6 Remarks

None.

5.7 Bit Patterns and Operations

5.7.1 General Description

Bit patterns are a common means of coding information in process control systems. Within a program, they are represented by strings of characters. Operations on bit patterns may be performed by the string operations of concatenation and substring extraction.

Functions are provided for conversion between strings and numeric values.

5.7.2 Syntax

1. string-supplied-function > BSTR$
2. numeric-supplied-function > BVAL

5.7.3 Examples

None

5.7.4 Semantics

The values of the supplied functions, as well as the number and types of their arguments, shall be as described below. B$ represents a string expression, V represents an index and R represents an integer constant whose value is 2, 8 or 16.

FUNCTION	VALUE
BVAL(B$, R)	The non-negative integer whose string representation is given by the string B$. R is the radix of the string representation of the value, eg: BVAL("101", 2) = 5 BVAL("2F", 16) = 47
BSTR$(V, R)	The string representation of the value of V, using radix R. Unless a fatal exception occurs, BSTR$ shall always return at least one character. In particular, the value of BSTR$ when V is zero is "0", eg: BSTR$(3.14, 2) = "11" BSTR$(15, 8) = "17"

The permissible characters that may appear in the string B$ depends on the value of R. If R is 2 the valid set is the digits 0 and 1. If R is 8 the valid set is the digits 0 to 7. If R is 16 the valid set is the digits 0 to 9 and the upper-case letters A to F.

5.7.5 Exceptions

The value of the string argument of BVAL is not a valid representation of a number in radix R (fatal).

The numeric interpretation of the value of the string argument of BVAL cannot be represented within the limits of the precision of numeric variables (fatal).

The numeric interpretation of the value of the string argument of BVAL exceeds the largest number representable (fatal).

The value of the first argument of BSTR$ is negative (fatal).

The value of the second argument of BVAL or BSTR$ is not 2, 8 or 16 (fatal).

5.7.6 Remarks

Typical uses for bit patterns are the manipulation of status registers, or of data from process objects in which individual bits represent specific objects such as switches or indicators.

6. Exception Handling

6.1 General Description

Exception handling facilities provide a means of regaining control of a program after an exception has occurred.

6.2 Syntax

```
 1. exception-handler            = handler-line block*
                                   end-handler-line
 2. handler-line                 = line-number HANDLER
                                   handler-name tail
 3. handler-name                 = routine-identifier
 4. end-handler-line             = line-number END HANDLER tail
 5. exit-handler-statement       = RESUME / RETRY / CONTINUE
 6. enable-handler-statement     = ENABLE HANDLER handler-name
                                   (comma RESUME AT line-number)?
 7. disable-handler-statement    = DISABLE HANDLER
 8. cause-statement              = CAUSE exception-type
 9. exception-type               = index
10. numeric-supplied-function > EXLINE / EXTYPE
```

A handler-name that occurs in an enable-handler-statement shall occur in some handler-line in the same program-unit. A given handler-name shall occur in at most one handler-line in a program-unit.

Exception-handlers shall not be nested within other exception-handlers or within def-blocks that do not constitute a program-unit.

Exit-handler-statements shall occur only within exception-handlers. The supplied-functions EXLINE and EXTYPE shall be invoked only within exception-handlers.

A control-statement shall not transfer control to a line within an exception-handler from outside the exception handler (other than to the first as the result of an exception), nor to a line outside an exception handler from a line within it.

6.3 Examples

Example 1: handling errors in input-replies by allowing the input-reply to be resupplied after issuing a suitable message

```
110 ENABLE HANDLER EXP1
120 PRINT "Enter your age and weight";
130 INPUT AGE, WEIGHT
140 IF AGE > 10 THEN
150    PRINT "What is your height in meters";
160    INPUT HEIGHT
170 END IF
```

```
300 HANDLER EXP1
310 PRINT "Please enter numbers only!"
320 RETRY
330 END HANDLER
```

Example 2: handling numeric overflows in a subprogram by setting a status return and exiting from the subprogram (other exceptions are handled by the default procedures)

```
100 SUB STATS (A(), M, S)
110     ENABLE HANDLER OFLO, RESUME AT 900
120     LET S = 0

800     HANDLER OFLO
810        IF EXTYPE = 1001 THEN
820           LET S = 1
830           RESUME
840     END IF
850     END HANDLER
900 END SUB
```

Example 3: handling a variety of exceptions arising in a single computation

```
100 ENABLE HANDLER OOPS
110 LET X = LOG(VAL(A$))
120 DISABLE HANDLER

800 HANDLER OOPS
810   SELECT EXTYPE
820   CASE 400:          ! A$ not numeric
830      CALL FIX(A$)
840      RETRY
850   CASE 3004          ! Bad argument for LOG
860      LET X1 = VAL(A$)
870      IF X1 = 0 THEN
880         LET X = -INF
890      ELSE
900         LET X = LOG(-X1)
910      END IF
920      CONTINUE
930   CASE ELSE
940      REM  Allow system to handle the exception
950   END SELECT
960 END HANDLER
```

6.4 Semantics

Execution of an enable-handler-statement shall enable the named exception-handler to process exceptions that subsequently arise during execution of the program-unit. At most one exception-handler shall be enabled at a time in a program-unit. If an enable-handler-statement is executed while an exception-handler is enabled, the currently enabled exception-handler shall be disabled and the named exception-handler enabled.

Execution of a disable-handler-statement shall disable the currently enabled exception-handler, if any such exception-handler exists.

When an exception occurs during the execution of a program-unit, the action taken shall depend upon whether an exception-handler is currently enabled in that program-unit. If no exception-handler is enabled, then the default exception-handling procedures specified in this Standard shall be applied. If an exception-handler is enabled, then the default exception-handling procedures, which require that the exception be reported, shall not be applied; instead, the enabled exception-handler shall be executed.

Within an exception-handler, the type of the exception that caused that handler to be executed shall be obtainable as the value of the parameterless function EXTYPE. The values of EXTYPE for all exceptions defined in this Standard are specified in Appendix 2. The line-number of the line whose execution caused the exception shall be obtainable as the value of the parameterless function EXLINE.

There are four means of exiting from an exception-handler. Execution of the exit-handler-statement CONTINUE shall cause execution to resume with the statement following the one that caused the exception. Execution of the exit-handler-statement RETRY shall result in the re-execution of the statement that caused the exception; if that statement was an input-statement, then the previous input-reply shall be discarded and a new one requested. Execution of the exit-handler-statement RESUME shall cause execution to resume at the line whose line-number was specified in the last-executed enable-handler-statement; if no line-number was specified in that statement, then execution shall resume at the line following the one that caused the exception. Execution of the end-handler-statement shall cause the exception to be handled by the default exception-handling procedures.

Execution of a cause-statement shall result in the occurrence of an exception of the specified type.

If an exception occurs during the execution of an exception-handler then that exception shall be handled by the default exception-handling procedures.

If a fatal exception occurs in a procedure that is a separate program-unit and no exception-handler is enabled there, or if the end-handler-statement is executed in the exception-handler invoked by that exception, then a fatal exception shall occur at the line that invoked the procedure. Such exceptions shall continue to occur until an invocation of a program-unit with an enabled exception-handler or the main-program is reached. If an exception-handler is invoked in this process, then the value returned by the EXTYPE function shall be 100000 plus the value that would have been supplied for EXTYPE in the program-unit in which the exception occurred. If the main-program is reached and no exception-handler is enabled there, then the exception shall be handled by the default exception-handling procedures specified in this Standard.

Lines in an exception-handler shall not be executed unless that handler is enabled and an exception occurs. If execution reaches the first line of an exception-handler in some other fashon, then it shall proceed to the line following the end-handler-line with no other effect.

6.5 Exceptions

An exception occurs during execution of an exception handler (fatal).

6.6 Remarks

The function EXLINE should be used with caution, as the use of editing facilities that renumber lines in a program may invalidate computations involving EXLINE. For example, the program fragment

```
1000 SELECT CASE INT(EXLINE/100)
1010 CASE 1, 2
     -
     -
1100 CASE 3 TO 7
     -
```

would probably behave differently if lines 100 to 800 were renumbered.

All positive values of EXTYPE are reserved for future versions of this Standard. Exceptions defined by local enhancements to this Standard should be identified by negative values for EXTYPE, following the categories established in Appendix 2. The value returned by EXTYPE for an exception defined in a local enhancement and occurring in a subprogram should be -100000 plus the negative value identifying the exception. For example, if an implementation chose an EXTYPE value of -4029 for an invalid parameter in a new built-in function, and if that exception occurred in a subprogram, but was not handled there, then the value of EXTYPE in an exception-handler in a calling program should be -104029.

It is recommended that implementations use the "zeroeth" value in a class of EXTYPE values to represent "other exceptions of this type". For example an EXTYPE value of 1000 might represent all overflows not defined in this Standard.

7. References

1. American National Standard for Minimal BASIC (1978) ANSI X3.60

2. ISO Minimal BASIC 1980 DIS 6376

3. ECMA-55 Minimal BASIC 1978

4. Hoare CAR Communicating Sequential Processes CACM 1978 Vol. 21 No. 8 pp. 666-677

5. Reference Manual for the Ada Programming Language United States Department of Defense July 1980

6. Hoare CAR Monitors: an Operating Systems Structuring Concept. Comm. ACM Vol. 17 No. 10 Oct. 1974, pp 549-557

Appendix 1. Distributed Systems and Independent Compilation

This Appendix is extracted from a EWICS TC2 working paper. It describes the current ideas on how to implement large or distributed systems. This Appendix does not form part of the proposed Standard. However, it is intended eventually to publish a supplement to the Standard defining an extension of IRTB for use in distributed applications.

INTRODUCTION

The Draft Standard for IRTB is oriented towards application configurations with common memory accessed by one or more processors, and for which the program is compiled as a unit. This Appendix describes an extension for use with application configurations comprising multiple processors without shared memory, or for large application programs for which it is desirable to divide the program into a number of separately compiled segments.

For independent compilation the problem is to define paths between ports that are used in different program units. The solution is to introduce a global declaration unit whose scope is all the programs relating to a particular application.

For distributed systems a facility must be provided for allocating activities and shared data sections to the various processors. When a program is divided into independently compiled units, it is convenient for these allocations to be defined in the global declaration unit.

The global declaration unit does not contain executable statements; its purpose is to define the structure of the application. The global unit has two parts: An intercommunication part that declares message paths between message ports and the visibility of shared data sections to shared-data ports, and a configuration part that declares the allocation of activities to processors and the association of physical process objects to specific process I/O ports.

Since message paths and shared-data access paths are declared outside the coding of the parallel sections, it is no longer necessary for connecting ports to have the same name. The modularity of the program is improved by allowing a parallel section to use local names for all its ports. An activity can then be reused or redistributed without changing its code.

The following paragraphs describe the global declaration unit and its relation to message ports, data ports and process ports in independently compiled programs. An implementation could use this information for compiler directives, as a command input to a preprocessor, or as a sort of JCL (Job Control Language) for the language processor. An alternative implementation would be to compile the declarations into tables that reside in computer memory and are used to resolve the linkages at program execution time, thereby allowing dynamic reconfiguration of the system while the program is running.

DATA STRUCTURES

Data structure declarations are necessary to allow the language processor to check the consistency of connected message ports, and to define the shared data. The declarations are as defined in section 5.3.

MESSAGE PATHS

The attributes of a message path are the names of the communicating activities, the local message-port names in each, the direction of data transfer, and the structure of the data. The syntax of a message path declaration is as follows:

```
1.  message-path-dec          = MESSAGE FROM section-name
                                message-port-name TO section-name
                                message-port-name
```

eg:

```
STRUCTURE REALS: 2 OF NUMERIC
MESSAGE FROM ALPHA MIX TO BETA NEXT OF REALS
```

The processor in which each activity runs is determined by configuration declarations (see below).

SHARED DATA

The declaration of data that is accessible to more than one concurrent activity is syntactically identical to the declaration of a data port defined in section 5.3. In addition the capability of mapping data ports onto the system shared-data is defined:

```
1.  data-mapping-dec          = ASSIGN section-name data-port-name
                                limits? TO shared-data-name limits?
2.  limits                    = left-parenthesis lower-bound colon
                                upper-bound (comma lower-bound colon
                                upper-bound)? right parenthesis
3.  lower-bound               = integer-constant
4.  upper-bound               = integer-constant
```

The integer-constant representing the lower-bound and upper-bound shall be unsigned. The upper-bound shall be larger than the lower-bound.

As an example of this feature consider a number of similar input processors with essentially the same program, which are collecting status information that must be available to a supervisor activity. The code for each input processor should not depend on which section of the system data it is supplying. Suppose ALPHA, BETA and GAMMA each supply 10 structures to a section of shared data 50 structures long called CHAN. Appropriate statements could be:

```
STRUCTURE BLOCK: 2 OF STRING, 4 OF NUMERIC
SHARED CHAN(49) OF BLOCK
     ASSIGN ALPHA MON(0:9) TO CHAN(0:9)
     ASSIGN BETA MON(0:9) TO CHAN(10:19)
     ASSIGN GAMMA MON(0:9) TO CHAN(20:29)
     ASSIGN SUP MON(0:49) TO CHAN(0:49)
```

where MON is the name of a shared-data port in each of the activities.

simple data items may be mapped onto simple data items or array elements, and vectors may be mapped onto sections of vectors or matrices. An alternative to the above example could be:

```
SHARED CHAN (9,4)
     ASSIGN ALPHA MON(0:9) TO CHAN(0:9, 0:0)
     ASSIGN BETA MON(0:9) TO CHAN(0:9, 1:1)
     ASSIGN GAMMA MON(0:9) TO CHAN(0:9, 2:2)
     ASSIGN SUP MON(0:49) TO CHAN (0:9, 0:4)
```

ALLOCATION OF ACTIVITIES TO PROCESSORS

Process peripherals are associated with a processor rather than with the activities currently running in it. To permit a real-time program to be independent of processor configurations, process type declarations are defined for use in the coding of the activities. Process paths and the mapping of process ports onto process paths are defined in the global declaration unit.

```
1. process-type-dec        = PROCESS qualifier process-port-name
2. allocation-section      = processor-block*
3. processor-block         = PROCESSOR processor-name processor-type
                             file-block*
4. file-block              = file-name activity-block*
5. activity-block          = ACTIVITY activity-list use-block
6. use-block               = (use-statement / process-mapping-dec)*
7. use-statement           = USE string-expression
8. process-mapping-dec     = ASSIGN process-port-name TO
                             process-path (comma process-port-name
                             TO process-path)*
```

Process type declarations are used instead of process-port-decs in real-time-programs.

Processor-name, processor-type and file-name are implementation-defined. Activity-list is a list of parallel section names. The string-expression in the use-statement identifies a file containing process-port-decs.

Assignments need not be made if the same names are used for process ports in the activities and process paths in the global declaration unit.

Examples of these statements are:

```
    PROCESSOR MONITOR LSI11
        FILE MONIP
            ACTIVITY ALPHA, BETA, GAMMA
                USE PRODEC
                ASSIGN FAIL TO LAMP1, TEMP TO THERM
        FILE MONOP
            ACTIVITY LOG
PROCESSOR DISPLAY APPLE
```

where PRODEC is the name of the file containing the process-port-decs for the processor MONITOR; FAIL and TEMP are the names of process ports in the activities ALPHA, BETA and GAMMA; and LAMP1 and THERM are the names of process-paths declared in the file PRODEC.

APPENDIX 2. Exception Codes

The following lists the values of the EXTYPE function corresponding to the exceptions specified in this document. The numbers in parentheses following each exception refer to the section in which that exception is specified. All these exceptions are fatal.

OVERFLOW __________ 1000

1008 Overflow in numeric value for process input (5.4).
1009 Overflow in numeric value from shared data (5.5).
1055 Overflow in string value for process input (5.4).
1056 Overflow in string value from shared data (5.5).

SUBSCRIPT ERRORS __________ 2000

2001 Subscript out of bounds (5.4, 5.5).

PARAMETER ERRORS __________ 4000

4201 String argument of BVAL is not a valid string in radix R (5.7).
4202 Numeric interpretation of the string argument of BVAL cannot be represented withi the precision limits (5.7).
4203 Numeric interpretation of the string argument of BVAL exceeds the largest number representable (5.7).
4204 The first argument of BSTR$ is negative (5.7).
4205 The second argument of BVAL or BSTR$ is not 2, 8 to 16 (5.7).

MATRIX ERRORS __________ 6000

6301 Mismatched dimensions for array in real-time structure (5.4, 5.5, 5.6).

INPUT/OUTPUT ERRORS __________ 8000

8105 Timeout during a process input or output operation (5.4).
8106 Timeout during a message send or receive operation (5.6).

REAL-TIME ERRORS __________ 12000

12001 Attempt to start an activity that is not stopped (5.2).
12002 Attempt to signal an event that has occurred, and has not yet restarted a waiting activity (5.2).
12003 Event reoccurs before it restarts a waiting activity (5.2).
12004 Illegal numeric value specified for time-expression (5.2).
12005 Illegal string value specified for time-expression (5.2).
12006 An event does not occur within the specified timeout interval (5.2).

APPENDIX 3. Implementation-defined features

A number of features referred to in this Standard have been left for definition by the implementor. The way these features are implemented shall be defined in the user or system manual for the implementation.

The following is a list of the implementation-defined features:

SECTION 5.1

Scheduling of parallel-sections.
Interpretation of the urgency of parallel-sections.
Where execution of a parallel-section can be interrupted.
Values of variables at the initiation of a parallel section.

SECTION 5.2

Which of several activities waiting for an event is restarted.

SECTION 5.3

Interpretation of the access-information for a process-port-dec.

SECTION 5.6

Which of several activities waiting to receive the same message shall actually receive it when the corresponding send-statement is executed.

GENERAL

It should be noted that implementation-defined features may cause a program to behave differently on different implementations, for the following and possibly for other reasons:

- The logical flow of a program may be affected by the algorithm used for the pseudo-random number sequence,

- The logical flow of a program may be affected by the value of machine infinitesimal and/or the value of machine infinity,

- The initial value of variables may affect the logical flow of a program that contains logical errors,

- The logical flow of a program may be affected by the order of evaluation of numeric-expressions,

- The behaviour of a program may be affected by the strategy of the implementation-defined scheduler.