IRTF

Industrial Real – Time FORTRAN

IRTF

Industrial Real – Time FORTRAN

Real Time FORTRAN Draft Standard

The European Workshop on Industrial Computer Systems (EWICS) has prepared a Draft for Standardization, which is available for public review and comment, on Industrial Real Time FORTRAN.

The Draft for Standardization was developed by EWICS Technical Committee 1 (EWICS TC1)in cooperation with Standards Committee 61 of the Instrument Society of America (ISA S61) and the American Technical Committee 1 of the International Purdue Workshop on Industrial Computer Systems (IPW TC1/A).

This Draft specifies a tasking model and a set of procedures to allow control of multitasking systems. Moreover it specifies procedures for binary pattern and bit processing, process input/output and the control of the access of shared files. The Draft for Standardization is based on FORTRAN 77. It is currently discussed to be submitted to ISO for international standardization.

Copies of the Draft may be obtained free of charge from the chairman of EWICS TC1:

Dr. Wilfried Kneis
Mergenthaler Linotype
Entwicklung

Frankfurter Allee 55-75
D-6236 Eschborn bei Frankfurt
Federal Republic Germany

Comments to the Draft received before August 31, 1981, will receive a written reply after being considered by the committee. Comments should be submitted to the chairman of EWICS TC1.

IPW/EWICS TC 1, 2.2/80 October 1980
(IRTF)

Draft Standard

Industrial Real-Time FORTRAN

Definition of Procedures for the Application of FORTRAN for the Control of Industrial Processes

Proposed by the
Technical Committee 1
of the International Purdue Workshop on Industrial Computer Systems
and of the European Workshop on Industrial Computer Systems (EWICS)

Note: This copy of the draft standard is being circulated for comment only. Please address all comments to the chairman of EWICS TC1:

Dr. Wilfried Kneis
Mergenthaler Linotype
Entwicklung

Frankfurter Allee 55 - 75
D-6236 Eschborn/Frankfurt
Federal Republic of Germany

The development of this standard has been supported by the Commission of the European Communities, Directorate-General III. The views expressed herein are, however, not necessarily those of the Commission.

FOREWORD

This Standard specifies a tasking model and a set of related routines to allow control of multi-tasking systems. The background to this work is given in ANNEX A, and suggestions and justification for some features are given in ANNEX B. ANNEX C deals with the problem of aborting tasks. All annexes are included for information purposes and are not part of the Standard.

No extensions or variations from this Standard should be implemented except for features explicitly declared in this Standard to be processor dependent.

The Standard has been prepared by the Technical Committee 1 of the European Workshop on Industrial Computer Systems (EWICS) in close cooperation with the ISA/S61 Committee and with the American Technical Committee 1 of the International Purdue Workshop on Industrial Computer Systems.

The standard was prepared by the following members of the EWICS committee:

C. G. F. Ampt	PTT	NL
W. Koblitz	Technische Universität Wien	A
C. Blume	Universität Karlsruhe	D
P. N. Clout	Los Alamos Scientific Laboratory, USA	GB+USA
A. J. Cox	University of Oxford	GB
G. Heller	Fachhochschule für Technik Mannheim	D
W. Kneis	Mergenthaler Linotype	D
W. Koblitz	Technische Universität Wien	A
K. Maliszewski	MERA - PIAP, Warszawa	PL
K. Mangold	AEG-Telefunken	D
O. Pettersen	Norwegian Inst. of Technology	N
U. Rembold	Universität Karlsruhe	D
D. A. Rutherford	University of Manchester	GB
J. A. M. Snoek	Delft University of Technology	NL
H. Sobiesiak	Kernforschungszentrum Karlsruhe	D
G. Teuschler	Siemens AG	D
M. Topschowsky	AEG-Telefunken	D
P. Urbainsky	Universität Erlangen-Nürnberg	D
A. J. H. Walter	Rutherford Laboratory, DIDCOT	GB
G. Wiesner	Hahn-Meitner-Inst. für Kernforschung	D

Many different people contributed to the development of this standard, but the influence by the American ISA S61 and IPW TC1-A Committees was particularly important. The members of these committees are:

A. Arthur	IBM Corporation
F. E. Bearden	The Cadre Corporation
R. H. Caro	Modular Computer Systems
L. M. Cartright	Inland Steel Corporation
R. L. Curtis	ALCOA
W. van Diehl	Hewlett Packard Company
M. R. Gordon-Clark	Scott Paper Company
M. N. Hands	Digital Equipment Corporation
C. C. Haskell	Union Carbide Company
W. Loper	Naval Oceans Systems Center
T. L. Luekens	Johnson Service Company
S. C. Schwarm	E. I. du Pont de Nemours
R. Signor	General Electric Company
R. E. Willard	Digital Equipment Corporation
D. W. Zobrist	ELDEC

CONTENTS

1 Scope and field of application

This Standard establishes external procedure references for use in industrial computer control systems. These external procedure references provide access to time and date information, permit interface of programs with executive systems, process input and output functions, allow manipulation of bit strings and provide a method for file handling.

These procedures are intended for use with programs written in FORTRAN conforming to the current ISO standard according to [16]. These programs are expected to be executable both in a solitary and in a multiprogramming environment under the control of a real-time executive system.

This standard is applicable to all FORTRAN systems which require multi-tasking features.

2 Definitions

Terms defined elsewhere in this vocabulary are underlined. They are, however, underlined only at the first occurrence within the same definition paragraph. Some of the definitions, star-marked in the table (*), are taken from ISO 2382/X.

access mode: The right or permission to access (read or write) a file granted by the processor following a request for such permission.

basic clock counts: Counts of the basic unit of the system real time clock as available for the user's program.

computation: A set of operations applied on a set of data, as available for the user's program.

critical region: A part of a sequential program operating on shared data such that this program part must have exclusive access to the shared data during the execution.

DORMANT: One of the definite states of a task. A dormant task is known to the executive system and is not in any of the states PENDING, RUNNING, or SUSPENDED.

event: A significant discrete occurrence or incident which is intended to affect some task execution in a planned manner. An event itself occurs instantaneously and sets an eventmark[1].

eventmark: Internal variable of the executive system, used to indicate that an event has occurred.
If the user's system contains parallel tasks, the eventmarks are shared data elements for the tasks.[1].

execution: The collection of actions performed by a computer processor carrying out instructions in a sequential manner.

executable program: A program including all of its functions and subroutines in a form suitable for execution.

executive routine;
executive system: The part of the processor which supports the procedures of this standard.

file: A collection of related records treated as a unit. For the purpose of this standard, the records are viewed as being of fixed length. Record storage and access are independent of the internal format of records.

initiation: The action taken by the executive system to start the execution of a task at its first executable statement.

multiprocessing*: A mode of operation that provides for parallel processing by two or more processors of a multiprocessor.

multiprogramming*: A mode of operation that provides for the interleaved execution of two or more computer programs by a single processor.

multitasking*: A multiple operation that provides for the concurrent performance or interleaved execution of two or more tasks.

NON-EXISTENT: One of the definite (formal) states of a task. A non-existent task is unknown to the executive system.

object task;
designated task;
referenced task: The task that is wanted or expected to be started, halted, stopped, or otherwise affected as a consequence of a system subroutine call.

operation: A deterministic rule for the generation of a finite set of data from another finite set of data.

overrun: Occurs when the condition for initiation of a task becomes true while the task is still running because of a previous initiation.

---

1) Supplementary information, see clause 4.3.1.

parallel tasks;
concurrent tasks: A set of tasks whose operations may overlap in time.

PENDING: One of the definite states of a task. A pending task has been associated with an event or time condition such that when the condition occurs, the task will be transferred to state RUNNING and initiated.

processor: The combination of a data processing system and the mechanism by which programs are transformed for use on that data processing system. (ANSI X3.9-1978.)

processor dependent: The action of the processor is not specified in this standard.

repetitive execution: Occurs when a task is repeatedly initiated, whether at fixed intervals or by repetitive events.

resourcemark: Internal variable of the executive system, used to indicate that a resource is exclusively reserved for a task.[2)]

RUNNING: One of the definite states of a task. A running task is executing in its virtual processor.

semaphore: A variable of the executive system, used for the exchange of synchronizing information between interacting parallel tasks. All semaphore operations in this standard imply critical region protection, provided by the executive system.

sequential order of operations: An order of operations that can only be performed strictly one after another in time without any overlap in time.

SUSPENDED: One of the definite states of a task. A suspended task has temporarily halted the execution of its virtual processor, and is waiting for a specified condition to continue the execution of its virtual processor.

task: A computation which can be scheduled.
The operations of this computation are performed in a strict sequential order of operations.
(See also definitions for object task and sequential order.)

time: absolute time:
Complete time and date specification.
relative time:
A time increment or difference.

virtual processor: An environment in which a task can run from the time it is initiated until it terminates without consideration of resource availability. A particular implementation serves to map a set of virtual processors onto a set of real processor(s). This mapping is processor dependent.

---

2) Supplementary information: See clause 4.3.2.

SECTION ONE

MULTIPROGRAMMING AND REAL-TIME FEATURES

This section describes several procedure references available for the user's program, and relating to multiprogramming and particularly real-time operation. For all calls of SECTION ONE the operation is generally considered indivisible, i.e. their operation shall behave as if they are not interrupted.

3 Date and Time information

For programming in a real-time environment, the user must have access to the time variables of the operating system. These time variables are obtained by system calls described in this clause.

Unambiguous time specification requires unique designation of time including complete date and an acknowledged calendar, defining time zero. Execution of reference to subroutine DATIM provides this complete information. The date refers to the Gregorian Calendar.

The calls are:

CALL DATIM(t1) For obtaining current date and time.

CALL CLOCK(j,k1,k2)
For obtaining the basic clock counts.

3.1 Obtain Date and Time

The form of this call is:

CALL DATIM(t1)

where:

t1 designates an integer array, into whose first 8 elements will be placed the absolute time, as expressed by the system's real-time clock at the time when the call is executed. These elements are as follows:

First element:		Counts of the basic clock.
Second	"	Milliseconds (0 to 999)
Third	"	Seconds (0 to 59)
Fourth	"	Minutes (0 to 59)
Fifth	"	Hours (0 to 23)
Sixth	"	Day (1 to 31)
Seventh	"	Month (1 to 12)
Eighth	"	Year

3.2 Obtain Clock Counts [1)]

Execution of a reference to this subroutine allows the user program to obtain the current value of the system real-time clock, expressed in basic clock counts.

The form of this call is:

CALL CLOCK(j,k1,k2)

where:

j designates an integer variable or integer array element into which the current value of the clock will be placed as a positive integer. j is counted up to a maximum value as given by k2, then set to zero and counted again.

k1 Number of basic clock counts per second. An integer value returned by the system. This argument shall be an integer variable or an integer array element.

k2 Specifies the maximum number j can attain. An integer value returned by the system. This argument shall be an integer variable or an integer array element.

4 General aspects of tasking

4.1 States and transitions

At any time, a task is in one and only one state. Actions executed by the executive system, other tasks, or the designated task itself, may cause transition from one state to another. These transitions are performed instantly, i.e. they are considered ideally to take no time.

This mathematical model of a task may be visualized by a "state diagram", like Fig. 1, in which the states are nodes, illustrated as circles, while transitions are drawn as pointed arrows from one node to another [11].

A multiprogramming system consists of several parallel tasks and can be considered as modelled by a number of disjoint but similar diagrams. It is feasible to apply a three-dimensional picture, with the similar diagrams sandwiched on top of each other and oriented so that the identical states of the individual diagrams cover each other.

State transitions are generally caused by subroutine calls, occurrence of events, or expiration of time limits. The name, form, and interpretation of the subroutine calls are standardized and described in the present document.

In the present document, a task is described by a mathematical model illustrated by the state diagram of Fig. 1. This model adheres to the following basic principles:

1. Transitions are non-ambiguous, i.e. for a given stimulus in a given state, the task can transit to only one possible new state.
2. Transitions are performed instantly, i.e. in zero time.
3. A task exists in one state only at a time.
4. The state model describes the behavior of tasks as seen by the application programmer.

---

1) Supplementary information, see Annex B.

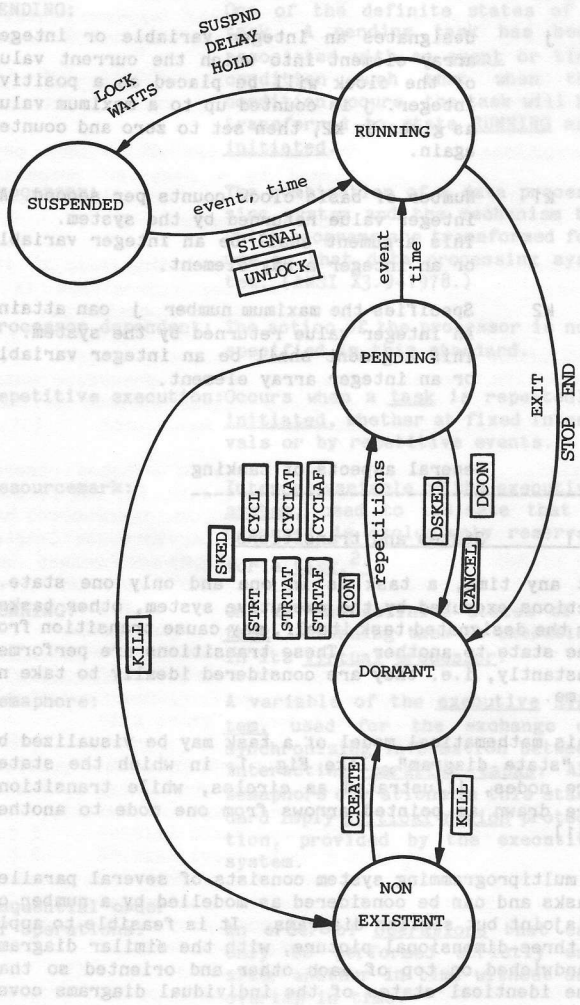

Figure 1. STATE MODEL AND TRANSITION DIAGRAM

The following symbolism is used for transitions in the state diagram:

- Capital letters in box: Effect on a task imposed by another task. I.e. a subroutine call in one task has the indicated effect on the designated task.
- Capital letters without box: Effect imposed by the task itself while in the state RUNNING.
- Small letters: Conditions under which the executive system performs the indicated state transitions.

With reference to principle 4 above, no attempt is made to describe executive system actions transparent to the application programmer. Consequently, the state RUNNING is related to the task's virtual processor: It is immaterial for the state of a task, whether a physical processor happens to be assigned to it, or the execution is temporarily hampered by the executive system due to limited availability of physical processors and the task's low priority. Thus, the model is as well adapted to multiprocessors as to single processor computers.

4.2 Multiple activation calls [1)]

Different tasks may issue apparently conflicting transition calls for the same object task. This will be the case during normal operation, as well as under error conditions, since the state of an object task is unknown at the time a transition call is made by another task.

Conforming to the requirement of one state only at a time, as listed in the previous clause, a distinction is made between state transitions and calls for such transitions. Transition calls are received by the executive system, which will apply its own scheduling strategy in handling such calls.

Depending on available resources, like internal table space etc. of the executive system, a transition call will be accepted or rejected. An argument will be returned after the reference, with appropriate value indicating whether the reference has been accepted normally or rejected.

4.3 Synchronization concepts

Within this standard, three concepts for the synchronization between tasks and the resolution of the resource contention are provided:

eventmarks,
resourcemarks,
semaphores.

Eventmarks and semaphores are mainly used for synchronization purposes, whereas the resourcemarks are mainly used for the resolution of resource contention. In the following subclauses these three concepts are described further.

Eventmarks, resourcemarks, and semaphores are local variables of the executive system and they may not be accessed other than by the mechanisms described in this standard.

4.3.1 Eventmarks

In the management of concurrent tasks, it is necessary to associate certain tasks with certain events. These events may be either external or internal events. External events are of some physical nature, like a contact closure, but the connection between an external event and its eventmark is beyond the scope of this standard. An internal event arises from specific program action (see description for CALL POST, clause 5.9.1).

1) Supplementary information, see Annex B.

Eventmarks are selected by reference to a numeric selector in the range 1 to n, where n is processor-dependent. Eventmarks have two states:

ON
OFF

The association of events to tasks is done by the subroutine calls CALL SKED (see clause 5.4), CALL CON (see clause 5.5.8), CALL SUSPND (see clause 5.7), and CALL HOLD (see clause 5.8.1).

An eventmark is turned to ON by the occurrence of an internal or external event. If one or more tasks are associated with this event, the executive system will cause each associated task to begin or continue execution. An eventmark is turned to OFF by direct program control or by the executive system as it services the tasks associated with the eventmark. Eventmarks can be changed only by the procedure references defined in this standard and by the executive system that services the events.

Eventmarks may also be set to the OFF state by a specific program action, the CALL CLEAR (see clause 5.9.2), and they may be masked and unmasked (see clauses 5.9.4 and 5.9.5). Additionally, the value of an eventmark may be tested by the logical function TESTEM (see clause 5.9.3).

4.3.2 Resourcemarks

A simple means to resolve contention for various resources is provided by the resourcemark concept, which permits one and only one task to use a resource at a time.

Resourcemarks are selected by reference to a numeric selector in the range 1 (one) to n, where n is processor dependent. Resourcemarks have two states:

LOCKED
UNLOCKED

This standard does not define what can be considered a resource. It is the responsibility of the user to associate a resource with a resourcemark. If the user wants to reserve a resource exclusively for the running task, he chooses a resourcemark for the resource and performs the reference CALL LOCK (see clause 5.10.1). Should the resourcemark be in state UNLOCKED at this moment, the resourcemark will change to state LOCKED, reserving the corresponding resource exclusively for this task. This reservation is later released by the task, by execution of a reference to EXIT (see clause 5.12) or UNLOCK (see clause 5.10.2).

Should a task attempt to LOCK a resourcemark which is already locked, the executive system will transfer this task into the state SUSPENDED, where it remains until the resourcemark becomes unlocked by some other task.

If several tasks are waiting for a resourcemark to be UNLOCKED, only one will be selected for execution by the executive system. (For details, see clause 5.10.2). This is a main difference from the eventmark concept, where all tasks waiting for an eventmark are transferred to state RUNNING if the eventmark is set to ON.

A resourcemark can also be set to LOCKED by reference to the logical function TLOCK (see clause 5.10.3).

4.3.3 Semaphores

Semaphores represent a synchronization concept useful for cases where a more advanced mechanism than those available by resourcemarks and eventmarks is desired. Unlike eventmarks and resourcemarks, a semaphore has an integer value, s . The value s of a semaphore must be initialized and may later be set by the reference CALL PRESEM (see clause 5.11.1).

If the task needs to wait until the value of a semaphore is greater than or equal to a specific value, the task uses the reference CALL WAITS (see clause 5.11.2). By execution of CALL WAITS, the executive system tests if the specified decrement j is greater than s. In this case, the calling task is transferred to state SUSPENDED, where it remains until the semaphore value s is incremented by another task to a value greater than or equal to j. If j is not greater than s , the value s is decremented by j and the calling task continues its execution.

By execution of CALL SIGNAL (see clause 5.11.3), the executive system increments by j the value s of the specified semaphore. This incrementation may bring a task, waiting for this semaphore, from state SUSPENDED to state RUNNING, if s becomes greater than or equal to j of the suspended task. If multiple tasks are waiting for this semaphore and are candidates for running, after the incrementation of s by j, the order in which these tasks transit to state RUNNING is processor dependent.

By setting s to certain values and choosing different values for j, a variety of advanced synchronization concepts and solutions for resource contention are possible.

In addition to the calls described above, the value of a semaphore may be read using the integer function IRDSEM (see clause 5.11.4).

5 Procedure references

5.1 Terms and summary of procedure references [1]

This clause contains a summary of the subroutine calls and function references described in subsequent clauses.

The following designations for parameters apply to several of the calls. If the exact meaning of these parameter designations deviates from what is described below, it will be marked specifically in the detailed description of the call. If the meaning is exactly as defined here, the description of the parameter will be omitted in the description of the call.

i specifies the task to be affected (object task). The argument shall be an integer array.
The contents of i may be partly generated by CREATE. i is used as input parameter in all other calls.

1) Supplementary information, see Annex B.

t,t1,t2 designate integer arrays, whose first 8 elements contain a specification of absolute or relative time. Negative values of elements are not permitted. These elements are as follows:

First element:		Basic clock counts
Second	"	Milliseconds
Third	"	Seconds
Fourth	"	Minutes
Fifth	"	Hours
Sixth	"	Day(s)
Seventh	"	Month(s)
Eighth	"	Year(s)

If, for absolute times, value 0 is used for one of the three date elements, this shall be interpreted as "current date", "current month", or "current year" by the executive system.

The interpretation of a relative time specification containing months or years different from zero is processor dependent.

m is set on return to the calling program, to indicate the disposition of the request as follows:

0 or less	:	undefined
1	:	request accepted
2 or greater	:	request rejected (error condition)

This argument shall be an integer variable or integer array element, local to the calling program. The processor may define specific values ≥ 2 to distinguish between certain reasons for rejection.

The list of function and subroutine calls, described in detail in subsequent clauses, is:

Full description in clause:	call:	parameters:
5.2	CALL CREATE(i,m)	i : identification of created task and associated program
5.3	CALL KILL(i,m) (opposite of CREATE)	
5.4	CALL SKED(i,s,e1,t1,t2,e2,m) (general scheduling)	s : mode selector e1: eventmark reference t1: absolute or relative time for first initiation t2: time period for cyclic initiations e2: reference to eventmark for overrun
5.5.1	CALL STRT(i,m) (start immediately)	
5.5.2	CALL STRTAF(i,t1,m) (start after time delay)	t1: time delay before initiation
5.5.3	CALL STRTAT(i,t1,m) (start at absolute time)	t1: absolute time for initiation
5.5.5	CALL CYCL(i,t2,m) (cyclic, with immediate first initiation)	t2: length of time interval
5.5.6	CALL CYCLAF(i,t1,t2,m) (cyclic, with delayed first initiation)	t1: time delay before first initiation t2: length of time interval
5.5.7	CALL CYCLAT(i,t1,t2,m) (cyclic, with absolute time spec. of first initiation)	t1: absolute time for first initiation t2: length of time interval
5.5.8	CALL CON(i,e,m) (establish event connection)	e : eventmark reference
5.6	CALL DSKED(i,s,e,m) (eliminate scheduling)	s : mode selector e : eventmark reference
5.6.1	CALL DCON(i,e,m) (eliminate event connection)	e : eventmark reference

5.6.2	CALL CANCEL(i,m) (eliminate time scheduling)	
5.7	CALL SUSPND(s,e,t,n,m) (suspend continuation of calling task for time period or until event)	s : mode selector e : reference to eventmark for end of delay t : time delay n : indicator for cause of end of delay
5.8.1	CALL HOLD(e,m) (suspend until event occurs)	e : reference to eventmark for end of delay
5.8.2	CALL DELAY(t,m) (suspend for a relative time)	t : time delay
5.9.1	CALL POST(e,m) (setting of an eventmark).	e : eventmark reference
5.9.2	CALL CLEAR(e,m) (resetting of an eventmark).	e : eventmark reference
5.9.3	TESTEM(e,m) (testing the state of an eventmark).	e : eventmark reference function value: condition of the eventmark
5.9.4	CALL MKEM(e,m) (setting the mask of an eventmark).	e : eventmark reference
5.9.5	CALL UNMKEM(e,m) (clearing the mask of an eventmark).	e : eventmark reference
5.10.1	CALL LOCK(r,m) (locking or a resourcemark).	r : resourcemark reference
5.10.2	CALL UNLOCK(r,m) (unlocking or a resourcemark).	r : resourcemark reference
5.10.3	TLOCK(r,m) (testing and locking of a resourcemark).	r : resourcemark reference function value: condition of the resourcemark
5.11.1	CALL PRESEM(r,s,m) (initialization of semaphore)	r : semaphore reference s : initial value of semaphore
5.11.2	CALL WAITS(r,j,m) (wait on semaphore)	r : semaphore reference j : decrement
5.11.3	CALL SIGNAL(r,j,m) (release semaphore)	r : semaphore reference j : increment
5.11.4	IRDSEM(r,m) (read semaphore value)	r : semaphore reference function value: value of semaphore variable
5.12	CALL EXIT (termination of execution).	

5.2 Creation of a new task

A new task is introduced to the real-time system by reference to subroutine CREATE. The designated task will be associated with some specified program, considered a resource like other resources, necessary for the task to perform. The associated program is normally assumed to exist in an executable form. Formally, and in terms of the state model, the task is transferred from NON-EXISTENT to DORMANT as effect of the reference (see fig. 1, clause 4.1).

A mechanism is assumed to exist outside the standard, to create and initiate at least a first task, i.e. the parent, which in its turn may create other tasks.

The form of the call is:

CALL CREATE(i,m)

where:

i specifies an integer array which contains all information necessary to specify the task and its associated program. The latter includes, among other items, its designation, where the program can be found such as description of file, residency while existent (primary memory resident or swappable), etc. The array may also contain the task's processor priority.

This array will in general also contain output information: references distinguishing this task from other tasks created from identical program code, reference to individual data sets, etc.
Such information may be used by other procedure references of clause 5. See clause 5.1, description of parameter i.
All details of this array are processor defined.

m see 5.1.

5.3. Eliminating a task from the real-time system[1)]

A reference to subroutine KILL will eliminate a designated task from the real-time system, by transferring it to state NON-EXISTENT. If the designated task is in state DORMANT or PENDING, the effect shall be carried out immediately. If the designated task is in state RUNNING or SUSPENDED, the termination shall affect only future executions. Thus the designated task will continue its present execution without any intervention by this call.

The form of the call is:

```
CALL KILL(i,m)
```

where:

i,m see 5.1.

5.4. Scheduling a task

Execution of a reference to the subroutine SKED, or its derivates as listed in clause 5.5, shall schedule the initiation of a designated task, establishing the condition for subsequent transition to RUNNING. If the designated task is in state DORMANT when the call is made, the designated task will transit to state PENDING. If it is in state PENDING already, the reference shall augment its conditions for subsequent transfer to RUNNING according to the arguments of the call, such that the designated task will transit to RUNNING when any of the conditions still valid, becomes true. The transition to state RUNNING requires the task being in state PENDING, otherwise an overrun condition occurs. The augmentation of running conditions is subject to any resource limitation of the processor, and any violation of such limitation will result in an error return.

When a task transits to RUNNING, the condition that caused this transition is removed from the possible complex of combined conditions, such that the other conditions remain. When a running task subsequently exits, by transition from RUNNING to DORMANT, possible scheduling conditions, remaining from previous scheduling calls shall cause immediate transition further to state PENDING. Note, however, the definition of the term "overrun" in clause 2 and the provision of its indication, as explained below.

Each normally accepted reference to the subroutine SKED causes the following effect:

After expiration of a specified time delay or at a desired absolute time or upon the occurrence of a specified event, the object task is transferred to state RUNNING and begins at the first executable statement of the program. The actual time resolution obtainable in a specific industrial computer system is subject to the resolution of that system's real time clock. If the object task is initiated by an event occurrence, the executive system will set the eventmark OFF, as part of the initiation. If more than one task is waiting for a specified eventmark to change to the condition ON, when the change occurs all these tasks will transit into state RUNNING.

The form of this procedure reference is:

```
CALL SKED(i,s,e1,t1,t2,e2,m)
```

where:

i,m see 5.1.

s is an integer expression, specifying three categories of task scheduling:

- the values of the argument s between 10 and 15 cause the task to be initiated once.
- the values of the argument s between 20 and 25 cause the task to be initiated periodically by time.
- the values of the argument s between 30 and 35 cause the task to be initiated whenever a specified eventmark becomes ON.

The values of the argument s between 10 and 15 define the first and only execution of the task.

= 10: start immediately
= 11: start at absolute time t1
= 12: start after time t1
= 13: start at event e1 (once)
= 14: start at absolute time t1 or event e1 (once)
= 15: start after time t1 or at event e1 (once)

The values of the argument s between 20 and 25 define the first execution of the task which is subsequently executed repetitively at the time period t2.

= 20: start immediately plus cyclic by t2
= 21: start at t1 plus cyclic by t2
= 22: start after t1 plus cyclic by t2
= 23: start at e1 plus cyclic by t2
= 24: start at t1 or e1 plus cyclic by t2
= 25: start after t1 or at e1 plus cyclic by t2

The values of the argument s between 30 and 35 define the first execution of the task which is subsequently executed whenever the eventmark specified by e1 becomes ON.

= 30: start immediately plus repeated by e1
= 31: start at t1 plus repeated by e1
= 32: start after t1 plus repeated by e1
= 33: start at e1 plus repeated by e1
= 34: start at t1 or e1 plus repeated by e1
= 35: start after t1 or at e1 plus repeated by e1

1) Supplementary information, see Annex B.

e1 specifies the eventmark for scheduling; an integer expression.

t1 designates an integer array containing the absolute, respectively relative, time for the scheduling. By relative time for scheduling is meant the time delay from the time the reference is executed until the intended running. (See 5.1.)

t2 integer array to designate the time period for cyclic runs. (See 5.1.)

e2 eventmark for overrun. Will be turned ON if overrun occurs. The action performed with the scheduled task is processor dependent.

5.5. Starting a task by simplified calls

The following calls for scheduling represent subsets of the features of CALL SKED. They are established for the ease of programming only:

```
CALL STRT(i,m)
     start immediately

CALL STRTAF(i,t1,m)
     start after time t1

CALL STRTAT(i,t1,m)
     start at time t1

CALL CYCL(i,t2,m)
     start immediately plus cyclic by t2

CALL CYCLAF(i,t1,t2,m)
     start after time t1 plus cyclic by t2

CALL CYCLAT(i,t1,t2,m)
     start at time t1 plus cyclic by t2

CALL CON(i,e1,m)
     start at event e1 plus repeated by e1.
```

5.5.1. Starting a task immediately

Execution of a reference to subroutine STRT establishes a condition for immediate transfer of the designated task to RUNNING via PENDING. Execution begins at the task's first executable statement. If in state DORMANT when the reference is made, the task will transit to state PENDING and continue immediately to state RUNNING.

The form of the call is:

```
CALL STRT(i,m)
```

where:

i,m see 5.1.

5.5.2. Starting a task after a specified time delay

Execution of a reference to subroutine STRTAF establishes a time delay as condition for transfer of the designated task to RUNNING via PENDING.

After expiration of a specified time delay after the time when the reference is executed, the designated task is expected to transfer to state RUNNING and will do so, if its state at that time is PENDING. If in state DORMANT when the reference is made, the task will transit to state PENDING. In state RUNNING, the task will begin at its first executable statement.

The form of the call is:

```
CALL STRTAF(i,t1,m)
```

where:

i,m see 5.1.

t1 designates an integer array, specifying the time delay after which the object task is to start its execution. (See 5.1.)

5.5.3. Starting a task at a specified absolute time

Execution of a reference to subroutine STRTAT establishes an absolute time as condition for transfer of the designated task to RUNNING via PENDING.

At the specified absolute time, the object task is expected to transfer to state RUNNING and will do so, if its state at that time is PENDING. If in state DORMANT when the reference is made, the task will transit to state PENDING. In state RUNNING, the task will begin at its first executable statement. The task is started immediately if the specified absolute time is already passed when the reference to STRTAT is executed.
The form of this call is:

```
CALL STRTAT(i,t1,m)
```

where:

i,m see 5.1.

t1 designates an integer array, specifying the absolute time at which the object task is to start its execution. (See 5.1.)

5.5.4 Starting a task in periodic execution

The calls for CYCL, CYCLAF, and CYCLAT for periodic execution have the following common features:

The designated task will be transferred to state PENDING if its present state is DORMANT or when it becomes DORMANT. Further, the reference establishes the condition for subsequent transfer of the designated task from state PENDING to RUNNING for a first execution and additionally causes future periodic executions. A reference to subroutine CYCL, CYCLAF, or CYCLAT has the same immediate effect as a reference for single execution; additionally, after its termination (e.g. by EXIT), the object task will be transferred immediately from state DORMANT to state PENDING for the next periodic execution, as indicated in Figure 1 by "repetitions". The next scheduled time is equal to the sum of the previous scheduled time and the interval specified by the reference to these subroutines. The re-scheduling under the said conditions shall continue until actively terminated by a call of subroutine CANCEL (see clause 5.6.2).

The actual running may be delayed unintentionally while in state RUNNING, because of running of other programs. Such delays will not be accumulated. If the execution is not finished before the time for next execution, an overrun situation exists and the action taken with the cycled task is processor dependent.

5.5.5 Initial start immediately

The form of the call is:

```
CALL CYCL(i,t2,m)
```

where:

i,m see 5.1.

t2 designates an integer array, specifying the nominal length of the time interval. (See 5.1.)

5.5.6 Initial start after a specified time delay

The form of the call is:

```
CALL CYCLAF(i,t1,t2,m)
```

where:

i,m see 5.1.

t1 designates an integer array, specifying a time delay for the initial activation, as measured from the time the call was made. (See 5.1.)

t2 designates an integer array, specifying the nominal length of the time interval. (See 5.1.)

5.5.7 Initial start at specified absolute time

The form of the call is:

```
CALL CYCLAT(i,t1,t2,m)
```

where:

i,m see 5.1.

t1 designates an integer array, specifying the absolute time at which the designated task is supposed to enter state RUNNING initially. This argument is exactly equivalent to parameter t1 of call for subroutine STRTAT. (See 5.1.)

t2 designates an integer array, specifying the nominal length of the time interval. (See 5.1.)

5.5.8 Connection of a task to an event

Execution of a reference to subroutine CON establishes a specified event as condition for transition from state PENDING to RUNNING. First, the task will transit to state PENDING if the state is DORMANT when the reference is executed, or, when the state becomes DORMANT. Then, if the related eventmark is or becomes ON, the designated task will transit from state PENDING to RUNNING and will begin with its first executable statement. The association between the event and the object task remains until actively cancelled by a reference to DSKED or DCON (see clauses 5.6 and 5.6.1).

The form of this call is:

```
CALL CON(i,e,m)
```

where:

i,m see 5.1.

e specifies the eventmark; an integer expression.

5.6. Elimination of previous scheduling

Execution of a reference to subroutine DSKED or its derivates, DCON and CANCEL, shall cancel specified scheduling conditions for an object task. Thus, it eliminates further effects from previous calls to subroutine SKED or its simplified derived subroutines.

If the object task is in state RUNNING or SUSPENDED, the cancelling shall affect only future executions. Thus, the object task will conclude its present execution, without any intervention by this call.

If the object task is in state PENDING, the cancelling will cause a transition to state DORMANT if no scheduling conditions remain.

The form of the call is:

```
CALL DSKED(i,s,e,m)
```

where:

i,m see 5.1.

s is an integer expression selecting one of the following conditions:
- =1: Eliminate all time based scheduling and scheduling by events as specified by argument e.
- =2: Eliminate all time based scheduling including repetition by time. Possible event connections remain unaltered.
- =3: Eliminate event-based scheduling, including event-based repetition, as specified by argument e. Possible time connections remain unaltered.

e specifies the eventmark(s), whose connection is to be eliminated:
- =-1: all eventmark connections for the designated task.
- =0: no event cancelling
- >0: reference to one specific eventmark, as specified by the value of e. This argument shall be an integer expression.

5.6.1. Elimination of event connections

Execution of a reference to subroutine DCON shall cancel any connection between an object task and a specified event. Thus, it eliminates further effects from a previous call to subroutine SKED or CON. If the object task is in state RUNNING or SUSPENDED, the cancelling shall affect only future executions. Thus, the object task will conclude its present execution, without any intervention by this call.

If the object task is in state PENDING, the cancelling will cause a transition to state DORMANT if no scheduling conditions remain.

The form of the call is:

```
CALL DCON(i,e,m)
```

where:

i,m see 5.1.

e specifies the eventmark; an integer expression. See 5.6.

5.6.2. Elimination of time connections

Execution of a reference to subroutine CANCEL shall cancel the future initiations of a designated, object task due to time scheduling by previous calls of SKED or its simplified versions. The executive system shall assure that no further move to the state PENDING shall take place due to previous cyclic scheduling.

If the object task is in any active state (RUNNING or SUSPENDED), the elimination shall affect only future executions. Thus, the object task will conclude its present execution, without any intervention by this call.

If the object task is in state PENDING, the cancelling will cause a transition to state DORMANT if no scheduling conditions remain.

The form of this call is:

```
CALL CANCEL(i,m)
```

where:

i,m see 5.1.

5.7 Delaying continuation of a task

Execution of a reference to the subroutine SUSPND shall provide a means whereby a running task is suspended (i.e. transits to state SUSPENDED) for a specified length of time or until a specified event has occurred. Then, the task shall transit back to state RUNNING and shall resume execution with the statement immediately following the call of subroutine SUSPND.

If more than one task is suspended and waiting for a specified eventmark to change to the condition ON, when the change occurs all tasks will transit to state RUNNING.

The time delay is defined as the nominal duration from the time when the call was made until the program resumes execution in its virtual processor, by being transferred to state RUNNING. The actual instants for the entering and leaving states RUNNING and SUSPENDED are subject to the resolution of the system's real-time clock and to the interrogating and activating actions performed by the executive system.
The form of the call is:

```
CALL SUSPND(s,e,t,n,m)
```

where:

s is an integer expression selecting the condition on which the suspension shall end:

= 1: at absolute time t
= 2: after time t
= 3: at event e
= 4: at absolute time t or event e
= 5: after time t or at event e

e specifies the eventmark for ending suspension, an integer expression.

t designates an integer array, specifying the absolute, respectively relative, time for the suspension. (See 5.1.)

n return parameter to indicate the cause of the end of the delay in case of s=4 or s=5:

= 1: end of delay by event e
= 2: end of delay by time t.

m see 5.1.

5.8 Suspending until event or time

The following calls for suspension represent subsets of the features of CALL SUSPND. They are established for the ease of programming only:

```
CALL HOLD(e,m)
  suspend until the event e has occurred

CALL DELAY(t,m)
  suspend for a time delay as specified by t.
```

5.8.1 Delay until event has occurred

Execution of a reference to subroutine HOLD shall suspend the calling task until a specified event has occurred. Then, the task shall transit to state RUNNING and thus resume execution with the statement immediately following the call of subroutine HOLD.
The form of the call is:

```
CALL HOLD(e,m)
```

where:

e specifies the eventmark for ending suspension; an integer expression.

m see 5.1.

5.8.2 Delay for a specified relative time

Execution of a reference to subroutine DELAY shall transit the calling task to state SUSPENDED for a specified duration. Then, the task shall transit back to state RUNNING, resuming execution with the statement immediately following the call of subroutine DELAY.
The form of the call is:

```
CALL DELAY(t,m)
```

where:

t designates an integer array, specifying the relative time for the suspension. (See 5.1.)

m see 5.1.

5.9. Eventmark operations

5.9.1. Setting an eventmark to the ON condition

Execution of a reference to subroutine POST shall set a designated eventmark to the ON condition. If the eventmark was already ON, there shall be no action. Because of an earlier transition call in some task, the ON condition may cause a task to be transferred from state PENDING or SUSPENDED to state RUNNING.

The form of this call is:

```
CALL POST(e,m)
```

where:

e specifies the eventmark; an integer expression.

m see 5.1.

5.9.2. Clearing an eventmark

Execution of a reference to subroutine CLEAR shall cause the designated eventmark to become OFF. If the eventmark was already OFF, there shall be no action.

The form of this call is:

```
CALL CLEAR(e,m)
```

where:

e specifies the eventmark; an integer expression.

m see 5.1.

5.9.3. Testing an eventmark condition

Execution of a reference to function TESTEM shall return a logical value TRUE if the specified eventmark was ON and a logical value FALSE if the eventmark was OFF. If the eventmark is unknown to the processor, a logical FALSE value will be returned, and the error parameter will indicate an error condition.

The form of this function reference is:

```
TESTEM(e,m)
```

where:

e specifies the eventmark; an integer expression.

m see 5.1.

5.9.4. Masking an eventmark

Execution of a reference to subroutine MKEM does not change the state of the designated eventmark but causes it to be masked. The masking effect is that the eventmark may freely change its state without any effect on tasks that might be pending or suspended waiting for this eventmark to be set.

The form of this call is:

```
CALL MKEM(e,m)
```

where:

e specifies the eventmark whose corresponding event is to be masked. The argument is an integer expression.

m see 5.1.

5.9.5. Unmasking an eventmark

Execution of a reference to the subroutine UNMKEM shall allow actions associated with the specified eventmark to be executed. If the eventmark is in the ON condition, then all actions associated with the specified eventmark will be executed.

The form of this call is:

```
CALL UNMKEM(e,m)
```

where:

e specifies the eventmark whose corresponding event is to be unmasked; an integer expression.

m see 5.1.

5.10 Resourcemark operations

5.10.1 Setting a resourcemark to the locked condition

Execution of a reference to the subroutine LOCK, shall cause the specified resourcemark to be locked. If the specified resourcemark was locked already, the executive system shall suspend the execution of the task. The form of the call is:

```
CALL LOCK(r,m)
```

where:

r specifies the resourcemark; an integer expression.

m see 5.1.

5.10.2 Setting a resourcemark to the unlocked condition

Execution of a reference to the subroutine UNLOCK shall cause one of the following actions:

a) If the resourcemark is unlocked, there will be no action.

b) If the resourcemark is locked and there are no tasks suspended as a result of a previously unsuccessful attempt to lock the associated resource, the resourcemark shall be unlocked.

c) If the resourcemark is locked and there are one or more tasks suspended as a result of previous attempts to lock the associated resourcemark, one and only one task shall transit to state RUNNING. The associated resourcemark remains locked. The criteria used to select the task to transit to state RUNNING are processor-dependent.

The form of the call is:

```
CALL UNLOCK(r,m)
```

where:

r specifies the resourcemark; an integer expression.

m see 5.1.

5.10.3 Testing and setting a resourcemark to the locked condition

Execution of the function TLOCK shall first test the specified resourcemark:
The function shall return the value TRUE if the resourcemark is unlocked; it shall return the value FALSE, however, if the resourcemark is locked. After this test, the resourcemark will be locked.

The form of this function reference is:

```
TLOCK(r,m)
```

where:

r specifies the resourcemark; an integer expression.

m see 5.1.

The reason for the use of TLOCK as compared to CALL LOCK is to allow a task to either reserve a resource, if it is unlocked, or to continue execution if it is locked. This would not be possible with CALL LOCK.

5.11 Semaphore operations [1)]

All semaphore variables have the form of local variables of the executive system, and the only means of access is through an argument, r , which refers to one particular semaphore. The value of a particular semaphore variable thus referred to is termed s in the following.

---

1) Supplementary information, see Annex B.

The effects of subroutines SIGNAL and WAITS are, respectively, the increasing and decreasing of the semaphore value by an amount j , a positive integer conveyed as the second argument of the calls. For WAITS, the decrease will only take place if the result is not less than zero; otherwise the calling task is suspended before the decrementing takes place, and continuation will not occur until after $s \geq j$, i.e. the decrementation linked to the continuation will yield a non-negative value.

The following clauses describe in detail the subroutine calls for the synchronizing mechanisms mentioned in the introduction above.

5.11.1 Initialization of semaphore

Execution of a reference to the subroutine PRESEM has two purposes:
Firstly, it declares intention of use of a particular semaphore, permitting the system to give diagnostic warnings at run time if another semaphore operation is issued referring to a semaphore that is not initialized. Secondly, PRESEM establishes the initial value for the semaphore. Normally, PRESEM is referenced only in the initialization phase of the execution of a real time program. The error parameter m will indicate an error condition if no semaphore exists with the indicated designation. The form of the call of PRESEM is:

```
CALL PRESEM(r,s,m)
```

where:

r specifies a semaphore. It is an integer expression.

s is the initial value given to the semaphore. Until the CALL PRESEM is executed for a particular semaphore, and the internal variable is assigned value s, the internal value is undefined, and another system call referring to this semaphore shall give an error return. This parameter is an integer expression.

A negative value is permissible. This is the only way a semaphore may attain a negative value. The effect of a negative initial value is, that a correspondingly greater increase by virtue of CALL SIGNAL is required before the releasing action can take place.

m see 5.1.

5.11.2 Wait on semaphore

Execution of a reference to subroutine WAITS will involve a possible suspension of the calling task, as controlled by the referenced semaphore.

By the end of the call, the semaphore value s will be reduced by the amount j . The reduction and subsequent continuation will only take place when this can be done giving s a non-negative value. Otherwise, the calling task is suspended until this decrementation can take place.

The form of the reference to WAITS is:

```
CALL WAITS(r,j,m)
```

where:

r see 5.11.1.

j is an integer expression, specifying the amount by which the semaphore variable is to be decremented, if applicable. The value of j must be positive (one or greater), and j=1 corresponds to the simple semaphore most commonly used.

m see 5.1.

5.11.3 Release of semaphore

Execution of a reference to the subroutine SIGNAL shall increment an integer semaphore variable, designated s in the expression below, and referred to by one of the arguments of the reference. The subroutine shall be granted exclusive access to this semaphore during its operation and will execute the following modification of its value:

s=s+j

where j is an argument, see below.
A change from zero or negative to positive value, caused by this operation, may provide a possible releasing transfer to state RUNNING for a task waiting in state SUSPENDED for this event to occur.

Other tasks, suspended in their execution of CALL WAITS relating to the same semaphore r, shall have their suspension condition re-evaluated after the present SIGNAL call is terminated. This will provide an opportunity for suspended tasks to be released and resume operation, as described above. This continuation is subject to all common restrictions pertaining to exclusive operation on a same semaphore. Thus, the effect will be that only one of the suspended tasks will be examined at a time. This examination may involve reduction of the semaphore value again, as a consequence of releasing operation or WAITS for the examined task. This examination continues as long as a possibility remains that the semaphore value is greater than or equal to the j-value of some suspended task. The order in which suspended tasks are checked otherwise is processor-dependent.
The form of the call is:

```
CALL SIGNAL(r,j,m)
```

where:

r see 5.11.1.

j is an integer expression, specifying the amount by which the semaphore variable is to be incremented, if applicable. The value of j must be positive (one or greater), and j=1 corresponds to the simple semaphore most commonly used.

m see 5.1.

5.11.4 Reading a semaphore value

A semaphore value may be interrogated by execution of a reference to the integer function IRDSEM. The purpose of this function is not that it be used as a synchronization operation, but only to provide a means to supervise synchronization in a system. Thus, a reference to function IRDSEM can, for example, provide information about how far a buffer or another shared resource is from saturation. When accepted (honoured), the reference will be granted exclusive access to the semaphore. If the semaphore is already being accessed by another system call when the reference to IRDSEM is made, the reference will be subject to the same deferred response and contention mechanisms as the other semaphore operations. On return, the function designator will have a value equal to the internal semaphore value when the reference was accepted.
The form of the reference is:

```
IRDSEM(r,m)
```

where:

r see 5.11.1.

m see 5.1.

5.12 Normal termination of execution

Execution of a reference to subroutine EXIT shall terminate the execution of a task and return the task to state DORMANT. Eventmarks and Semaphores shall not be affected. Resourcemarks previously locked by this task shall be unlocked (see 4.3.2), and files released.

The form of this call is:

```
CALL EXIT
```

The common FORTRAN operations STOP and END provide alternative means to terminate execution of a task. However, the effect on files and other resources described in this standard is processor dependent.

SECTION TWO

BINARY PATTERN AND BIT PROCESSING

6 Introduction

The function references described in this section provide mechanisms for operations on bit patterns as well as individual bits of the internal representation of integer varables.

The operations presuppose that integer numbers are considered as if they are unsigned integers. The arithmetic shift operation (clause 7.2.2) is an exception.

With respect to arguments referencing individual bits, it is assumed a bit numbering rule in which bit number 0, the rightmost bit, is the least signifcant bit.

7 Binary Pattern Processing

7.1 Boolean operations

Boolean operations provided are: OR, AND, EOR, and NOT. These operations are implemented as integer functions. The implicit type for OR, AND and EOR is indicated by the use of I as the first letter of the function name. Their parameters, j and k, are integer expressions.

After execution of the functions, the parameters remain unchanged. The operations are performed on corresponding (equally numbered) bits of the two operands, giving the corresponding bit value of the result. The values indicated in the following truth tables represent the individual and corresponding bits of the arguments and of the function value.

7.1.1 Inclusive OR

The form of this function is:

IOR (j,k)

The value of the function is computed from the values of the parameters j and k according to the following truth table:

j	0	1	0	1
k	0	0	1	1
Function Value	0	1	1	1

7.1.2. Boolean AND

The form of this function is:

IAND (j,k)

The value of the function is computed from the values of the parameters j and k according to the following truth table:

j	0	1	0	1
k	0	0	1	1
Function Value	0	0	0	1

7.1.3. Boolean Complement

The form of this function is:

NOT (j)

The value of the function is the logical complement of the parameter value, j, according to the following truth table:

j	0	1
Function Value	1	0

7.1.4. Exclusive OR

The form of this function is:

IEOR (j,k)

The value of the function is computed from the values of the parameters j and k according to the following truth table:

j	0	1	0	1
k	0	0	1	1
Function Value	0	1	1	0

7.2. Shift Operations

The shift operations provided are logical, arithmetic and circular. The shift operations are implemented as integer functions. The functions have two parameters, j and n, considered as integer expressions.

j specifies the value (binary pattern) to be shifted

n specifies the shift count:

>0	indicates a left shift
=0	indicates no shift
<0	indicates a right shift

If the absolute value of the shift count is greater than the number of bits in a numeric storage unit, then the result is undefined.

The parameter values are not changed by the shift operations.

7.2.1. Logical Shift

The form of this function is:

```
ISHL (j,n)
```

All bits representing the parameter j are shifted n places. Bits shifted out from the left end or the right end, as the case may be, are lost. Zeros are shifted in from the opposite end.

7.2.2. Arithmetic Shift

The form of this function is:

```
ISHA (j,n)
```

Argument j and the function value are considered as signed integers. All bits representing the parameter j are shifted n places. In the case of a right shift (n<0), zeros are shifted into the left end if j is positive, and ones are shifted in if j is negative. The bits shifted out of the left end are lost. In case of a left shift (n>0), zeros are shifted into the right end while the bits shifted out of the left end are lost. In a left shift an arithmetic overflow may occur.

7.2.3. Circular Shift

The form of this function is:

```
ISHC (j,n)
```

All bits representing the parameter j are shifted circularly n places; i.e., the bits shifted out of one end are shifted into the opposite end. No bits are lost.

Note: The number of bits representing j is processor dependent.

8 Bit Processing

Individual bits of an integer can be tested with the functions for bit processing. The functions have two parameters j and k, which are integer expressions.

j specifies the binary pattern

k specifies the selected bit, numbered as in clause 6.

If k is negative or greater than the number of bits that are used to represent an integer value, the result of the function is undefined.

The parameter values are not changed by these functions.

8.1. Bit Testing

The form of this function is:

```
BTEST (j,k)
```

This function is of type LOGICAL. The kth bit of parameter j is tested. If it is 1, the value of the function is TRUE ; if it is 0, the value of the function is FALSE.

8.2. Set Bit

The form of this integer function is:

```
IBSET (j,k)
```

The value of the function is equal to the value of parameter j with the k'th bit set to 1.

8.3. Clear Bit

The form of this integer function is:

```
IBCLR (j,k)
```

The value of the function is equal to the value of parameter j with the k'th bit set to 0.

8.4. Change Bit

The form of this integer function is:

```
IBCHNG (j,k)
```

The value of the function is equal to the value of parameter j with the k'th bit complemented.

SECTION THREE

PROCESS-INPUT/OUTPUT

9 Introduction

The user of Industrial Real-Time FORTRAN must be able to address specific process devices for his application. As the majority of Input/Output (I/O) systems are computer dependent, this can only be standardized in a universal way by standardized calls to driver routines which are especially written for each I/O system. Computer independent I/O systems are either standardized, or standards are under consideration (CAMAC, GPIB, MEDIA, etc.). Standardized I/O calls designed for these systems are equally valid to the calls presented here but are outside the scope of this standard.

10 Scope of the process I/O and general structure of the I/O routines

The process peripheral is the link between the process, or its terminal devices, and the central processing unit of the computer. Data describing the space and time behavior of the process are received by a processing unit and prepared so that they can be transferred to the central processing unit through an I/O interface. The variety of tasks required by the process has resulted in a large number of peripheral devices from the different manufacturers. However, in the course of many years of hardware development, largely compatible and generally accepted lines of development have become established. They may be characterized by the following statements.

I/O ports are distinguished by their individual addresses. The I/O port designations (addresses) used in the procedure references will probably in most systems be identical to the individual hardware addresses, but this is not mandatory. This relation is considered a processor dependent feature beyond the scope of this standard.

The FORTRAN standard specifies that one statement must be completed before the processing of the next statement begins. The standard process I/O here described adheres to this rule. The calling task will wait for completion and this operating mode is indicated by the last letter W (waiting) of all subroutine names.

The following designations for parameters apply to several of the referenoes. If the exact meaning of these parameter designations deviates from what is described below, it will be marked specifically in the detailed description of the reference. If the meaning is exactly as defined here, the description of the parameter will be omitted in the description of the reference.

The procedures for process I/O normally have four (in a special case, five) parameters which in the following will be designated in the general form with i, j, k, m (and n if needed). Such an I/O call has the general form:

```
CALL procio(i, j, k, m)
```

where:

procio indicates one of the subroutines subsequently described.

i specifies the number of values to be transferred, an integer expression.

j specifies the name of an integer array or array element that contains necessary information for description of the I/O ports, i.e. address and data conversion information. The orderly representation of information is processor dependent.

k specifies the name of an integer array or array element that contains the input or output values.

m Status indicator. Its value characterizes the "success" of a call:

≤ 0	undefined
=1	all data have been transferred
≥ 2	error conditions

This argument shall be an integer variable or an integer array element.

11 Input/Output of analog values

For input we distinguish between hardware implemented sequential and random input. In the first case, for sequential input, the input parameter j contains the address of the first analog input; further addresses will be generated automatically. In the second case, the full sequence of addresses must be given in the array j. For output, the form is always random, i. e., all addresses are given in array j.

Generally, the format of j is system dependent. See annex B8.

The relationship between the range of an input or output port and its corresponding element in k is processor dependent. Some suggested design guidelines may be found in annex B8.

11.1 Sequential analog data input

Execution of a reference to the subroutine AISQW reads a sequence of analog input ports of sequential addresses.
The form of the call is:

```
CALL AISQW (i, j, k, m)
```

where:

i specifies the number of analog input ports to be read. The parameter is an integer expression.

j is a description of either hardware or software information for the acquisition and for the conversion of the first and the following analog ports. It is the name of

an integer array or of an element. See annex B8.

k array for recording the converted analog values. It is the name of an integer array or of an element. See annex B8.

m see 10.

11.2 Analog data input in random sequence

The subroutine AIRDW reads a sequence of analog input ports in a specified order.
The form of the call is:

```
CALL AIRDW (i, j, k, m)
```

where:

i specifies the number of input ports to be read. The parameter is an integer expression.

j is a description of the hardware and software information for the acquisition and for the conversion of each analog value. It is the name of an integer array. See annex B8.

k array for recording the converted analog values. See annex B8.

m see 10.

11.3 Analog data output

The subroutine AOW outputs a sequence of analog values to a collection of analog output ports in a specified order.
The form of the call is:

```
CALL AOW (i, j, k, m)
```

where:

i specifies the number of analog output ports, an integer expression.

j contains information for the data conversion and transfer. It is the name of an integer array. See annex B8.

k array from which the analog values are output. It is the name of an integer array. See annex B8.

m see 10.

12 Input/Output of digital values

For this type of input/output it is assumed that, while the effective information may be represented at times by a single bit, it will, nevertheless, be necessary to transfer digital values (considered as entities using whole numeric storage units or words) into or out of an integer array (for example 16 bits for each numeric storage unit).

12.1 Digital input

The form of the call is:

```
CALL DIW (i, j, k, m)
```

where:

i specifies the number of digital values input, an integer expression.

j contains hardware and in some case software information for conversion and transfer. It is the name of an integer array. A possible reset specification can also be contained in j.

k array in which the digital values will be stored. It is the name of an integer array.

m see 10.

12.2 Digital output

For the output, pulse output (Digital Output Momentary) is distinguished from digital output with a permanently held value (Digital Output Latching).

12.2.1 Digital pulse output

Execution of a reference to the subroutine DOMW performs pulsed output to a collection of I/O ports. A pulse will occur on those bits selected by a binary 1 of the corresponding bit and element of an array k. No pulse appears on bits selected by binary 0. The pulse duration is indicated in a suitable form, by parameter n.
The form of the call is:

```
CALL DOMW (i, j, k, n, m)
```

where:

i specifies the number of digital values output, an integer expression.

j contains hardware information for transfer of each digital value. This parameter is the name of an integer array.

k array representing the digital values to be output. It is the name of an integer array. The first outputted value will be taken from the first array element (i.e. the element whose index is 1.

n number of time units of the computer clock for the pulse duration. If the processor does not allow selection of duration, this argument is ignored but must be present. This argument shall be an integer expression.

m see 10.

12.2.2 Latched digital output

For latched digital output (DOLW), in addition to the output field, a mask field is also required to indicate which bits are to be changed in the output. The parameter k is therefore subdivided into k1 and k2.
The form of the call is:

```
CALL DOLW(i, j, k1, k2, m)
```

where:

i	specifies the number of digital words, an integer expression.
j	contains hardware information for every digital value that is output. This parameter is the name of an integer array.
k1	array representing the digital values to be output. The parameter is the name of an integer array.
k2	designates an array whose values define digital outputs which can be changed by the subroutine. A bit set in the k2 array indicates that the digital output will be changed to the state defined by the corresponding bit position in the corresponding integer array element in k1. The order of the elements in k1 and k2 will correspond to the order in j. This argument shall be an integer array name, or an integer array element.
m	see 10.

SECTION FOUR

FILE HANDLING

13 Introduction

These external procedure references provide means for controlling the access of files, and also provide means for resolving problems of file access contention in a multitasking/multiprocessing environment. In such an environment, it is expected that concurrent tasks will attempt to access the same file at the same time; therefore, the external procedure references defined here provide the information necessary for the processor to resolve such simultaneous access in an orderly manner. The method for resolution of access control is left to the processor.

The procedure references in this section are intended to provide the methods by which the task can inform the processor of the manner in which it intends to use the file, but the references are not intended to require specific properties or attributes to be associated with the referenced files. They provide the means to avoid contention problems when used in conjunction with sound program design but the implementation of this standard is no assurance that such problems will not arise.

The terminology used in this section is defined in clause 1 and in other references, primarily in the document describing Standard FORTRAN ANSI X3.9-1978, [16].

14 Background Information

Files exist in most computing systems and can have various attributes and features, such as:

- A file can contain data, programs, or catalogue information.
- There can be a variety of ways for file access such as sequential, direct, and stream.
- A file can be created or deleted by a task, by a system utility, or at system generation time.
- A file can have security attributes associated with the file for the purpose of ensuring file privacy.
- When a file is associated with a task, this association can be restricted by the processor for reasons of privacy.
- A file can be associated with a set of related concurrent tasks and this association can be restricted to assure orderly resolution of contention problems among the concurrent tasks.
- A file can be internal or external to a task.
- A file can reside on fixed or removable media.
- A file can reside on main storage or backing storage.
- Restrictions for reasons of privacy or contention may apply to a file or a component of a file such as records and data items.

15 File System Environment

In industrial real time computer systems, concurrent task operation with shared resources such as files is a common occurrence. This standard does not address all the areas of file management but is concerned with the problems that most commonly arise in industrial real time computer systems.

The following table shows those features covered by the standard and those excluded; however, the excluded features may affect the result of a request for association of a concurrent task to a file. Such restrictions on association are processor dependent and are outside the scope of this standard.

FEATURES AND ATTRIBUTES OF FILES

Included in the Standard	Excluded from the Standard
- Files whose contents are considered to be data.	- Files whose contents are not considered to be data by the accessing concurrent task.
- Files which exist on fixed media only or on removable media that are not removed.	- Files that exist on removable media which are removed.
- Files that reside in main storage or in backing storage.	
- Files that are external to a concurrent task.	- Files that are internal to a concurrent task.
- Creation and deletion of files by a concurrent task.	- Creation and deletion of files by a system utility or at system generation.
- The association of a file to a concurrent task for both system created and for concurrent task created files.	- Methods of file access.
- Restrictions on file access as applied to the file.	- Restrictions on file access as applied to a component of a file.
- The association of a file to a concurrent task irrespective of the method of access (e.g. direct, sequential, or stream).	- Attributes of a file for the purpose of ensuring file privacy.

16 Procedures to control file access

16.1 Introduction

The procedure references defined in this section of the standard are non-interruptible; that is, the processor will only execute one such procedure reference at a time. This requirement ensures that the features described are executed in an orderly manner.

The argument m, shown below, shall be set equal to or greater than two (2) in value when the request is not accepted by the executive system. Individual implementation may specify unique values of m within the allowable range to designate the specific reason for which the request was rejected.

16.2 Creation of files

Execution of a reference to the subroutine CFILW shall establish, but not open, a named file. Files established by CFILW do not have any privacy attribute to restrict a concurrent task from accessing the files. The contents of a newly created file are undefined by the standard. The form of call is:

```
CALL CFILW(j,n1,n2,m)
```

where:

j specifies the file.
The argument is either:
a) an integer expression
or b) an integer array name
or c) a procedure name
or d) a character expression
The processor shall define which of the above four forms are acceptable.

n1 specifies the number of storage units per record in this file. This argument shall be an integer expression.

n2 specifies the maximum number of records in this file. This argument shall be an integer expression.

m is set on return to the calling task to indicate the disposition of the request. The value must be 1 or greater.
1 - File successfully created
2 or greater - File not created
This argument shall be an integer variable name or integer array element name.

16.3 Deletion of Files

Execution of a reference to the subroutine DFILW shall remove a file from the file system. Any file created by the mechanism of clause 16.2 can be deleted by the execution of a reference to DFILW, but deletion will not be effected if the file is currently open to any task. The form of this call is:

```
CALL DFILW(j,m)
```

where:

j specifies the file.
The argument is either:
a) an integer expression
or b) an integer array name
or c) a procedure name
or d) a character expression
The processor shall define which of the above four forms are acceptable.

m is set on return to the calling task to indicate the disposition of the request. The value must be 1 or greater.
1 - File successfully deleted
2 or greater - File not deleted
This argument shall be an integer variable name or integer array element name.

16.4 Opening Files

Execution of a reference to the subroutine OPENW shall associate the unit specified by the task with the named file, and shall define the desired access mode of that task to the file. The form of this call is:

```
CALL OPENW(i,j,k,m)
```

where:

i specifies the unit by which the file, named by the argument j, is referenced in the task. This argument shall be an integer expression.

j specifies the file.
The argument is either:
a) an integer expression
or b) an integer array name
or c) a procedure name
or d) a character expression
The processor shall define which of the above four forms are acceptable.

k specifies access mode desired by the task. It is a declaration of the task's intended use of the file. This argument shall be an integer expression.
The following values are defined:
1. Unlocked - Read/write or write-access is requested by the calling task; other tasks are also allowed the same access.
2. Protected read - Read access is requested by the calling task and allowed to other tasks.
3. Locked - Read/write or write-access is requested by the calling task; The calling task excludes any file access by other tasks.

m is set on return to the calling task to indicate the disposition of the request. The value must be 1 or greater.
1 - File successfully opened to the calling task.
2 or greater - File not opened to the calling task.
This argument shall be an integer variable name or integer array element name.

Limitations

If the file is currently open to another task, the disposition of a possible request for a particular access mode will be as follows:

Unlocked Fails if another task currently has the file open in the Locked or Protected modes; otherwise succeeds.

Protected read Fails if another task currently has the file open in the Locked or Unlocked modes.

Locked Fails.

Any attempt to open a file will be successful only if the file exists. If the file was created by a mechanism outside of the standard, the attributes given to the file at its creation may restrict the granting of an access mode to the task.

16.5 Closing Files

Execution of a reference to the subroutine CLOSEW shall end the task association of the specified logical unit with a named file. The form of the call is:

```
CALL CLOSEW(i,m)
```

where:

i specifies the unit. The argument shall be an integer expression.

m is set on return to the calling task to indicate the disposition of the request. The value must be 1 or greater.
1 - File successfully closed to the calling task.
2 or greater - Non-performance
This argument shall be an integer variable name or integer array element name.

16.6 Modify Access Mode

Execution of a reference to the subroutine MODAPW shall change the calling task's access mode of a file previously opened by the calling task without closing and reopening the file.

If the calling task does not have access mode to the file, the request fails.

If the request for change cannot be granted, the previous access mode remains in force. The form of this call is:

CALL MODAPW(i,k,m)

where:

i specifies the unit. This argument shall be an integer expression.

k specifies the new access mode desired. This argument shall be an integer expression.
The following values are defined:
1. Unlocked - Read/write or write-access is requested by the calling task; other tasks are also allowed the same access.
2. Protected read - Read access is requested by the calling task and allowed to other tasks.
3. Locked - Read/write or write-access is requested by the calling task; The calling task excludes any file access by other tasks.

m is set on return to the calling task to indicate the disposition of the request. The value must be 1 or greater.
1 - Access mode requested is granted to the task
2 or greater - Access mode before the request remains in force.
This argument shall be an integer variable name or integer array element name.

Limitations

If the file is currently open to another task, the request for a change of mode will fail.

If the file was created by a mechanism outside of the standard, the attributes given to the file at its creation may restrict the granting of an access mode to the task.

ANNEX A

HISTORICAL BACKGROUND

A.1 Introduction

The FORTRAN language was originally an IBM-development in 1955. Since that time FORTRAN has become the most widely used high level language for scientific applications, and large powerful user libraries are available in FORTRAN.

FORTRAN was proposed for standardization in 1967 and was finally standardized internationally in 1972 by ISO. The current definition of FORTRAN is contained within ref. [16].

A.2 Special requirements

The wide use and the proved excellence of FORTRAN for scientific applications soon led to its use for industrial real-time systems. These applications require special operations, i.e. real-time operations, bit-string manipulation and facilities for process-I/O. These operations can only be accomplished in one of the following two ways:

(1) FORTRAN remains the basic language for arithmetic calculations and for reading and writing of data by standard peripherals. The special operations are realized by elements outside the syntax of FORTRAN.

(2) The complete real-time language remains within the syntax of FORTRAN including the special operations. This means that these operations have to be FORTRAN-subroutines or FORTRAN-functions.

The first approach leads to typical real-time languages which offer the user simple but powerful programming facilities for the special operations. In developing such extensions the designers try to apply all the features of the real-time operating system used; thus these extensions become dependent on the actual system. This was especially true for the early developments of this kind (see e.g. [2], [3], [4]). PROCOL, as a later development, avoids this disadvantage; this language was created in France for a series of French computers. It offers very advanced features for real-time programming (see e.g. [5]).

All languages with extensions outside the syntax of FORTRAN need special compilers for their translation.

In choosing the approach (2), where the language remains within the syntactical frame of FORTRAN, one gains the advantage that a first compilation and check of the user program can be done on any computer for which a FORTRAN compiler exists. On the other hand this approach, using subroutines and functions, leads to somewhat clumsy handling of the added CALLs and FUNCTIONs and their associated parameters. Yet this may be considered as a minor inconvenience as the many users of FORTRAN are well accustomed to this kind of programming.

Industrial real-time FORTRAN has also to be compared with two other families of real-time languages: industrial real-time BASIC and languages specifically designed for real-time applications, like PEARL, RTL/2, etc.

The industrial real-time BASIC languages are very easy to learn and to apply. They are well suited for simple and small problems. They can be implemented relatively easily in large as well as in small computer systems.

The languages of the specific real-time type deliver very powerful programming features to the user. These languages are therefore well suited to large and complex problems. Their implementation (compiler and real-time system) is relatively expensive.

The industrial real-time FORTRAN languages implemented by approach (1) above are often similar to the specific real-time languages. On the other hand the industrial real-time FORTRAN languages implemented by approach (2) are in many respects between BASIC and the specific real-time language type. Thus, this language offers the user an alternative to these two language facilities. Consequently, a language according to approach (2) has to be relative simple; this means that the number and complexity of the additional operations have to remain restricted because of the ease of learning and programming.

A.3 Source of the Standard

In order to prevent the development of many incompatible real-time languages, T. J. Williams and others in 1970 founded the "Workshop on Standardization of Industrial Computer Languages" at Purdue University. After a union with another institution in 1973, the workshop was named "International Purdue Workshop on Industrial Computer Systems". The "FORTRAN Committee" of the Purdue Workshop was very active and successful from the beginning. For various and good reasons this committee chose approach (2) above with all special operations being kept within the syntactical frame of the FORTRAN language. The committee developed in a relatively short time a first proposal for real-time FORTRAN which was approved and published by the ISA as ISA Standard S61.1 (1972) [6]. The paper contains the following groups and numbers of special operations:

- 3 CALLs and the FORTRAN statement STOP for controlling the state of concurrently activated "programs" [1].
- 6 CALLs for process-I/O
- 5 INTEGER FUNCTIONs for bit-string manipulation applied on an INTEGER used as a bit-string.

---

1) In this paper the term "task" is used instead of the term "program" used in [6], [7], [8], and [9]. See clause 2.

A second supplementary paper ISA S61.2 (1973) was published one year later. This paper contains the following groups and numbers of special operations:

- 7 CALLs for handling random unformatted files
- 1 LOGICAL FUNCTION for testing an individual bit in an INTEGER
- 2 CALLs for setting and clearing of an individual bit in an INTEGER

These ISA standards have gained great acknowledgement and, by worldwide use, great importance (the proposal for file handling excepted). The early development and the publication of these standards by the American "FORTRAN Committee" of the Purdue Workshop and the ISA has been an important step for industrial process control.

In the meantime, the ISA draft standards were developed further and were finally approved as ANSI/ISA standards. (Thus, ANSI/ISA S61.1 [8] contains all sections of the old papers except the file handling; the file handling is the subject of ANSI/ISA S61.2 (1978) [9].) As a significant difference from earlier papers, the 1976 version of S61.1 does not allow subroutine calls without wait on the completion of the procedure.

As the paper ISA S61.1 contains some procedure references which have gained wide acceptance, this paper can be considered as a "Basic Industrial Real Time FORTRAN" of the Purdue Workshop. It contains only 3 CALLs and the statement STOP for the management of parallel tasks (= programs). Since more extensive mechanisms are needed, the American "FORTRAN Committee" of the Purdue Workshop has worked intensively on a supplementary paper about the management of parallel tasks, organized as ISA S61.3 [15]. Parallel to this work, the working group "Prozess-FORTRAN" of the German VDI/VDE has developed a complete proposal "Prozess-FORTRAN 75", published as a draft standard of the VDI/VDE [10].

"Prozess-FORTRAN 75" comprises 11 CALLs for the management of parallel tasks and thus offers a restricted but powerful tool for programming real-time operations. The binary pattern and bit processing is similar to ISA S61.1 (1976); merely the description is different.) There are additional INTEGER FUNCTIONs for arithmetic and circular shift and a CALL for a bit change. The process I/O is practically identical to ISA S61.1 (1976); only the standardization of the analog I/O is performed in the direction of a restrictive standard. The file handling is different from the ISA S61.2 (1977) standard (the latter being in a proposed state at that time).

In 1976 the Purdue Europe[1] Technical Committee 1 on Industrial Real-time FORTRAN was founded by members of the VDI/VDE working group "Prozess-FORTRAN" and other specialists in Europe. The committee decided to work out a complete proposal for industrial real-time FORTRAN.

Up to the end of 1977, the cooperation between the American and European committees was rather loose, by only one joint meeting per year at the International Purdue Workshop meeting. Thus, the American and European committees worked rather independently of each other, with the result that the papers produced were quite different in several respects. Therefore, the ISO TC97 SC5 WG1, the Purdue Europe Workshop, the ECMA TC8 and the International Purdue Workshop all requested the American and European committees to develop proposals largely, or preferably completely, identical. In order to reach this objective, cooperation had to be intensified considerably. Since the middle of 1978, nearly all American meetings were attended by 1 or 2 Europeans and vice versa. Since this close cooperation was established, the American and European committees have made significant progress in reaching agreements. Thus, a quite good compromise could be reached on a majority of the diverging points. The achievement of a complete agreement and a common American/European proposal is accomplished by this standard.

This Industrial Real-Time FORTRAN standard will need to be adapted from time to time to allow the progress of FORTRAN and also the progress of real-time operating systems. Thus supplements must be expected.

ANNEX B

FURTHER DESCRIPTIONS PERTAINING TO INDIVIDUAL CLAUSES

B.1 Definitions and naming rules

Some of the definitions listed in clause 2 are inspired by or borrowed from such literature as [12], [13], and other sources. The definitions correspond largely to those commonly used in the International Purdue Workshop and are identical or nearly identical to definitions of the ISA standard S61.2 [9] and the proposed ISA standard S61.3. [15]. Definitions of the states refer to Figure 1 of Section One and its description in Clause 4 and may be clearer if the figure and its description are consulted.

The names of the calls in this standard have been chosen according to a few rules:

1. The names should be descriptive - giving an indication of their use.
2. They should be different from calls in any of the ISA standards S61.1, S61.2 and S61.3, if such a call is not completely identical to a call in the other standards. In the latter case, an identical name is desirable, and only then.
3. The names should not be too common, otherwise they might easily coincide with names naturally selected by users for certain application programs.

Rule number 2 has the desired effect, that an implementator may choose to keep calls already implemented according to an older standard, and existing user programs will not become obsolete just because of the new implementation.

1) Earlier designation "Purdue Europe", now EWICS.

B.2 Use of CALL CLOCK (clause 3.2)

The call CALL CLOCK is useful, first of all for applications requiring a somewhat lower level of programming. All three parameters are output arguments, j (the first in line) supplying the instantaneous value of the system clock. This value directly represents the contents of the counting register of the clock.

The next two arguments supply system constants necessary and sufficient to achieve system independence in the use of this call, to allow the user to preserve portability and system independence for his programs.

The first of these arguments, k1, supplies the basic clock frequency, whereas the last argument informs about the modulo of the clock, i.e. it reveals the number of bits of the clock register. The modulo is equal to the value of k2 plus one. One clock tick after the value of j is equal to k2, j will be reset to zero and start counting upwards again.

B.3 Choice of task model (clause 4)

Commonly, in many implementations, a call to the executive system (by calls of SKED, STRTAF, STRTAT, etc.) for state transition to PENDING causes listing in a table, appropriately called "pendingtable". This is a queue of transition calls waiting to be effected, i.e. intended to cause state transitions some time later. All entries in this table involve some conditions for future activations, and these conditions are either time dependent (STRTAF, STRTAT) or event dependent (CON). Entries with time conditions will generally be sorted according to the activation time. Often, executive system implementors will find it convenient to split this queue into two: a time queue and an event queue. With an ordered time queue, the executive system's task of monitoring when time is due for any task will be simple: it has only to check the first element, since this always represents the next task to be activated. The other part of the "pending-table" contains entries made by reference to the subroutine CON. All these elements refer to events as their conditions. The executive system will check this queue when external interrupts are received, as well as at some other instances, in order to be able to react on possible programmed events, for example eventmarks set to ON by reference to subroutine POST.
Similar tables may exist to direct the supervision of suspended tasks; in fact, often these two sets of tables will be combined.

Transition calls like those mentioned will cause entry in the "pending-table" up to the point where the maximum table space is exhausted. When this occurs, the transition call will be rejected with the error parameter (termed m in clause 5) indicating this fact by being set to a value greater than one.

The obvious benefit of a system like this will be that a call for a state transition will be independent of what state the object task happens to be in at the time the call is made. Therefore, there will be no conflict with the state, and error returns are less likely to occur. A "minimum" system without such buffering will behave like the system described when the pending table is full. Since this may happen for any table of finite size, the programming of the tasks must allow for this by checking the error parameter after any activation call. The benefit of a system with pending table comes, however, not so much from simplification of the application program, as from the substantial time savings obtained from the reduced number of error returns.

A further advantage of buffered calls is that multiple activations are possible, yielding an OR-function of activations of a certain task. For example, an operator action to schedule a certain task to start its operation in five minutes does not conflict with a previous call for operation at 10.00 o'clock next day.

This rule will also give a simple and logical operation of the CYCL, CYCLAF, CYCLAT, and CON calls, i.e. the calls causing possibility of multiple subsequent activations (RUNs): Whenever a task attains state DORMANT, (by CALL EXIT or STOP), the pending-table is checked. If listed, the task will be transferred immediately to state PENDING, guided by an OR-function of run-conditions from the table-entries. If the activating call was one of the recurrence calls CYCL, etc., or CON, the table condition will remain. The same will be the case if multiple calls of STRTAF or STRTAT were made, leaving the future activation times in the table. Thus, there will be no difference, in principle, between the immediate effects of multiple calls of STRTAF, etc. and the recurrence calls CYCL, etc. and con. The effect of the recurrence calls will only be that the condition in the table entry will be adjusted recursively at the instant when the task after a CALL EXIT is transferred past DORMANT to PENDING.

What is said above about the specific calls like STARTAT, CYCLAT, etc. pertains also to references to the comprehensive CALL SKED, with appropriate argument values.

A different issue is the problem caused by tasks whose execution is excessively delayed, so that the time or other RUNNING conditions for their next execution, as scheduled by call of SKED or it derived simplified versions, is already satisfied before EXIT of the previous run is called. Such conditions are termed "overrun" and should normally be regarded as errors and give some error reaction. The problem is, however, that the error condition does not exist when the scheduling call is made, and most systems do not keep any record of connections back to the scheduling task; at least, such operations would hardly conform to FORTRAN. Thus, in general there is no obvious receiver of such error reactions within the user's program. Most appropriately, such error reactions should be handled by the system, and as such, it is outside the scope of this standard. The standard provides one possibility, however, by which the user may make his own mechanism for handling overrun conditions. By using the CALL SKED, rather than the simplified calls, this call has a return argument, e2, for indication of overrun. This eventmark will be turned ON in case of overrun. Thus, a special user-supplied task may be connected to this eventmark by CALL CON and thus initiated upon the overrun condition, thereby monitoring overrun conditions in other tasks.

B.4 Use of error parameter (clause 5.1)

The processor may define specific values for $m \geq 2$ to distinguish between certain reasons for rejection. For example, it could be advantageous, after a rejected

reference to subroutine WAITS, to get some further information. This could be supplied by the following range of values of m:

m is set on return to the calling task, to indicate the disposition of the request as follows:

0 or less	: undefined.
1	: request accepted, synchronization obtained.
2	: request rejected, because of non-existing semaphore.
3	: request rejected, because j is outside allowed range.
4 or greater	: request rejected, because of unspecified error.

This argument shall be an integer variable or integer array element.

B.5 Creation of a new task (clause 5.2)

In that the tasking model used in this standard implies independent tasking, a reference to subroutine CREATE shall not form a dependent relationship between the creating task and the created task. That is, the termination of one task should not cause a state transition of another task, in spite of the fact that the one task may have created the other task.

It is not the intent of this standard, however, to preclude alternative subroutines, similar to CREATE, which do form independent tasking relationships.

B.6 Use of CALL KILL (clause 5.3)

The processor may impose certain restrictions regarding which calling tasks may be permitted to eliminate other tasks. For example, a reference to subroutine KILL may be effective only if the designated task has been created by the same calling task, i.e. there exists a parent - child relationship.

B.7 Use of semaphores (clause 5.11)

Semaphores represent one of the most basic synchronization mechanisms, and this is probably also the one most widely accepted. This consideration has been dominant for the selection as the main synchronization mechanism included in the proposed standard.

Semaphores are traditionally manipulated by Dijkstra's P and V primitives. [14]. Although these primitives are widely known by these terms, single letter identifiers should be avoided, since they can too easily be confused with user defined names. Also, the semaphores and semaphore operations contained here are more advanced than the common simple type, since increment is not confined to value 1. Thus, the semaphore manipulating subroutine calls are described here, using the designations WAITS (WAIT on Semaphore), and SIGNAL, corresponding to P and V respectively.

Argument r specifies a semaphore. This means, that r=2 specifies one semaphore, r=3 another, completely different, semaphore. In a user's program, r will usually be specified as a constant.

Exclusive access rights of a task referencing WAITS are released when the task is suspended. This is necessary to avoid deadlock, and since this suspension is a normal and intended operation of the synchronization mechanism under control of the operating system, it does not conflict with the basic atomic requirement mentioned before.

B.8 Termination of execution (clause 5.12)

It is recommended to terminate execution of a task by call of subroutine EXIT, which performs a defined release of the resources Eventmarks, Semaphores, and Resourcemarks. This is not the case for the common FORTRAN operations STOP and END, whose effect on these resouces is processor dependent.

B.9 Binary pattern operations (clauses 6 and 7)

The arithmetic in most application programs of process technology is much less characterized by the processing of INTEGER, REAL, or DOUBLE PRECISION quantities -- as, for example, in engineering-scientific programs -- than by the processing of binary patterns or single bits, which often represent some form of status values or packed data. Since this standard complies with the rules of the general ISO FORTRAN standard, it is not possible to introduce new data types for the elements: binary pattern and bit. If one proceeds, however, on the premise that in most modern digital computers integer values are represented by their binary values, then one can with the aid of standardized procedure references effectively process binary patterns as well as single bits.

In the procedure references described in section two it is assumed that integer numbers are represented in binary form. The representation of negative values is processor dependent.
For example the following internal representation would be obtained with a word length of 16 bits on a two's complement machine:

Value	Binary Pattern															
Bit:	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
-1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
10	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0
-10	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0

The bits are thus numbered from right to left.

This standard does not prohibit implementations of the binary pattern operations as intrinsic functions.

B.10 Analog input operations (clauses 10 - 11)

The parameters j and k for analog I/O have the following meaning:

The parameter j consists of a measurement specification and a measurement point address.

The measurement specification may for example contain the measurement range of the analog input unit. The measurement point address specifies the address(es) of the analog I/O points; this also comprises a channel address, if required. Generally, the format of j is system dependent.

For block transfers of sequential analog inputs, only one element of j is required. This may be determined by one or two integers.

For block transfers of random analog inputs or outputs, the measurement point address at least changes from point to point. Therefore, a description field for j with several elements is required.

The relationship between the measurement range of an input port and its corresponding element in k is processor dependent.

The following scaling rules have the advantage that the engineering units y can be computed from the converted values x with the same formula y=f(x) for all computers with the same integer representation without any knowledge of the special coding of the ADC used.

- The analog value zero is represented by the numerical value zero.
- A unipolar positive analog signal is represented by a positive numerical value such that 100 % of the measurement range is represented by half of (the maximum value +1) of integer variables. This allows a measurement overrange of approximately 99 %. Thus, in the case of 16 bit storage unit for integer variables, 100 % of the measurement range would correspond to 2**14 = 16384 and the representation accuracy would be 2**(-15) or about .003 % of the measurement range.
- Bipolar, as well as negative unipolar, analog signals are represented similarly to unipolar positive signals, such that negative analog values are represented by negative numerical values in the usual format of the processor.

ANNEX C

ABORTION OF TASKS

An "aborting" mechanism is sometimes needed in an industrial RT system. There is, however, several valid arguments against the inclusion of a mechanism, capable of forceful terminating another task which may be in an arbitrary state at that moment. Such a mechanism will involve the risk for leaving the computer and its memory, files, and other resources in a messy, indefinite, and possibly dangerous state. Abortion, or forceful termination of an object task, may be necessary because of malfunction in a process section, or some other unusual situation. The abortion is effected by a call, issued in some supervising task, often running interactively in operator dialog.

This annex outlines a safe method, by which the application programmer is not given a general aborting tool, but a prescription on how to program an equivalent mechanism that can be used safely and in an ordered manner.

A task will often have sections where abortion would be improper, or fatal. Equally, there are sections, where a possible abort could be done without problems. The application programmer is the obvious person most able to identify these sections.

Assume an eventmark is defined to be used to guide an "abort" for a certain task. For reference, let us denote this eventmark "IABORT" in the following. This eventmark may be set ON by a supervising task, possibly following an operator request. In all tasks, where the application programmer decides that abort may be necessary and can be allowed, he includes the following and similar statements at the "safe spots" of the program:

```
 90 IF (TESTEM(IABORT,M)) GO TO 9000
    ....

9000   CONTINUE
C      HERE MAY BE INCLUDED ANY PROGRAM PORTION
C      DEEMED NECESSARY OR CONVENIENT BY
C      THE APPLICATION PROGRAMMER, TO BE
C      EXECUTED JUST BEFORE THE FORCED TERMINATION.

       CALL EXIT
       END
```

A supervising task may now abort the object task by calling:

```
       ....
       ....
       ....
       CALL CANCEL( i ,M1)
       IF (M1.NE.1) GO TO 8010
       CALL POST(IABORT,M1)
       IF (M1.NE.1) GO TO 8000
       ....
       ....
```

The effect is that the aborted task itself may perform a "graceful" termination, taking care of opened files, dangerous process states, etc.

Remark, that CANCEL and POST should be called in that order. Otherwise, a slight possibility exists, that a new execution of task i will be initiated between call of POST and CANCEL.

If the task to be aborted is in state SUSPENDED, it generally does not generate any harmful operations. It may remain there forever, though, waiting for a condition that will never occur, because of some malfunction. In this case, this task may occupy some resources that are needed by the rest of the system. Usually, the operating system is provided with some mechanism to deallocate such resources, so this is not covered any further here. For the purpose of taking care of the case that the object task is PENDING, the

aborting statements include a CALL CANCEL in the controlling task, whereby the object task transits back to DORMANT immediately.

ANNEX D

FILE HANDLING

This document is based on the programming language FORTRAN (ISO 1539 - 1980) [16]. In section four, file handling, the access to file is assumed to be performed by the standard FORTRAN input/output statements.

ANNEX E

REFERENCES

[1] ISO Recommendation R 1539, Programming Language FORTRAN. ISO, Geneva (Switzerland), 1972.

[2] Hohmeyer, R. E.: CDC 1700 FORTRAN for Process Control. IEEE Transactions on IECI, Vol. 15, No. 2, Dec. 1968.

[3] INDAC 8. Digital Equipment Corporation, 1969.

[4] DDP-516 OLERT, Programmer's Reference Manual. Honeywell Inc., Doc. No. 1300 72069 A, EM-602, 1969.

[5] Bonnard, P.: PROCOL: A Programming System Adapted to Process Control. IFAC Congress, Paris, 1972.

[6] ISA S61.1 (1972) Standard. Industrial Computer System FORTRAN Procedures for Ececutive Functions and Process Input-Output. Instrument Society of America, 1972.

[7] ISA S61.2 (1973) Draft Standard: Industrial Computer System FORTRAN Procedures for Handling Random Unformatted Files, Bit Manipulation, and Date and Time Information. Instrument Society of America, 1973.

[8] ANS/ISA S61.1 (1976) Standard. Industrial Computer System FORTRAN Procedures for Executive Functions, Process Input-Output, and Bit Manipulation. Instrument Society of America, 1976.

[9] ANSI/ISA S61.2 (1978) Standard: Industrial Computer System FORTRAN Procedures for File Access and the Control of File Contention. Instrument Society of America, 1978.

[10] VDI/VDE 3556, Prozess-FORTRAN 75, eine Erweiterung von FORTRAN für Prozessrechner-Anwendungen. VDI/VDE-Richlinie (Entwurf). VDI/VDE - Gesellschaft für Mess- und Regelungstechnik, Düsseldorf, 1978.

[11] O. Pettersen: Management of Parallel Activities in Real-Time FORTRAN. State Model and State Transitions. 2nd edition. Minutes of the Purdue Europe Spring Meeting 1977 at ISPRA CCR Euratom, Vol. II. IRIA, Rocquencourt, 1977.

[12] P. Brinch Hansen: Operating System Principles. Prentice-Hall, 1973.

[13] A. C. Shaw: The Logical Design of Operating Systems. Prentice-Hall, 1974.

[14] E. W. Dijkstra: Cooperating Sequential Processes. In "Programming Languages" (V. Genuys, ed.), pp 43 - 112. American Press, New York, 1968.

[15] ISA S61.3 (Aug. 1978) Draft Standard: Industrial Computer System FORTRAN Procedures for the Management of Independent Interrelated Tasks. Instrument Society of America, 1978.

[16] International Standard ISO 1539 - Programming Languages: FORTRAN (ISO 1539 - 1980)