Burroughs

B 5000/B 6000/B 7000 Series

Work Flow Language

REFERENCE MANUAL

RELATIVE TO MARK 3.2 RELEASE

PRICED ITEM

Printed in U.S.A.

MARCH 1981

5011794

Burroughs

B 5000/B 6000/B 7000 Series

Work Flow Language

REFERENCE MANUAL

RELATIVE TO MARK 3.2 RELEASE

PRICED ITEM

Printed in U.S.A.

MARCH 1981

5011794

Correspondence regarding this document should be addressed directly to Burroughs Corporation, P.O. Box 4040, El Monte, California 91734, Attn: Publications Department, TIO–West.

Burroughs

PUBLICATION CHANGE NOTICE

PCN No.: 5011794-001 Date: March 1, 1982

Publication Title: B 5000/B 6000/B 7000 Series WFL Reference Manual (March 1981)

Other Affected Publications:

Supersedes: N/A

Description:

Incorporates the Mark 3.3 System Software release.

Replace these pages		Add these pages
TOC-1	6-51	5-1-B
TOC-3	6-55	5-10-A
2-9	6-57	5-11B
5-1	6-59	5-11D
5-3	A-3	6-1-B
5-5	A-5	6-10-A
5-7	A-7	6-24-A
5-9	A-9	6-39-B
5-11	A-11	
5-19	Index-1	
5-29	Index-3	
6-1	Index-5	
6-23	Index-7	
6-37		
6-39		

Printed in U.S. America

B 5000/B 6000/B 7000 Series WFL Reference Manual

TABLE OF CONTENTS

1. INTRODUCTION

The Work Flow Language (WFL) provides the user with a means of constructing a job that allows tasks or programs to be run on B 5000/B 6000/B 7000 Series computer systems. Each command and facility available through WFL exists either to simplify or amplify this one basic function. WFL allows each job to be described as a set of interrelated tasks to be performed. Both serial and parallel execution are possible through WFL.

WFL accepts jobs from a large variety of sources, such as the following:

On-line card readers.
Remote Job Entry (RJE) stations.
Operator Display Terminals (ODTs).
Load control tapes.
START and WFL statements from CANDE sessions.
START statements from running WFL jobs.
File and array ZIP statements.

With a few exceptions, identical jobs may be presented to WFL from all of the above sources.

The WFL compiler is a true compiler because it produces code to control the tasks in a job as prescribed. Thus, all user jobs are "programs" written in WFL. The WFL compiler performs the following functions:

1. Syntactically checks the accepted WFL statement input.

2. Generates code to handle the tasks that constitute the job as specified by the WFL statements.

3. Generates the jobfile disk file for the job.

WFL, as used in this document, refers to "new" (post-2.9) WFL. "Old" WFL indicates that form of WFL used prior to the 2.9 system release.

The following documents contain related information.

B 7000/B 6000 Series I/O Subsystem Reference Manual, form 5001779.
B 7000/B 6000 Series System Software Operational Guide, Volume 1, form 5011661.
B 5000/B 6000/B 7000 Series System Software Operational Guide, Volume 2, form 5011679.
B 6700/B 7700 On-Line Maintenance and Test (MAT) Language Information Manual, form 5000169.

SCOPE

This reference manual is divided into the following six sections and two appendices:

Section 1 INTRODUCTION

This section describes the purpose of WFL and explains the scope and organization of this manual. The continued availability of old (pre-2.9) WFL is discussed, and the Work Flow Management (WFM) system is briefly described. The syntax and components of railroad diagrams are described.

Section 2 BASIC ELEMENTS AND CONSTRUCTS

The basic elements used in WFL are syntactically defined in this section.

Section 3 DECLARATIONS

Allowed declarations are described in this section.

Section 4 EXPRESSIONS

Expressions that may be formed using WFL variables are listed and defined in this section.

Section 5 JOB STRUCTURE

The structure of the WFL job is presented and discussed in this section. The components of the job are described.

Section 6 STATEMENTS

Each WFL statement is syntactically represented and explained in this section. Library maintenance concepts are also presented and discussed.

Appendix A TASK ATTRIBUTES AND MNEMONICS

The various <task attribute>s and <task mnemonic>s are described in this appendix.

Appendix B WFL CONTROL OPTIONS

Two WFL control options (ERRORLIMIT and INCLUDE) are discussed in this section.

CONTINUED USE OF THE "OLD" WFL LANGUAGE

The new WFL described in this document includes changes that have been made since the 2.9 release. Many of these modifications and extensions cause WFL decks containing old (pre-2.9) WFL statements to become invalid. Unless old WFL is explicitly specified, all references in this manual are to the new WFL. The current WFL compiler runs job decks written in either new or old WFL, but no job deck is accepted which contains statements from both new and old WFL.

All new WFL jobs begin with the pair of tokens BEGIN JOB. Old WFL jobs begin with either JOB, USER, CLASS, or a specific statement starter (not BEGIN). The WFL compiler immediately detects whether the job is of the old or new form.

The WFL compiler accepts jobs of either the old or the new job format from all possible job submission sources without modification of jobs or operating procedures.

Jobs keyed in at an ODT (via the CONTROLLER) which do not explicitly begin with JOB or BEGIN JOB are automatically prefixed with "JOB;" or "BEGIN JOB;" depending on the setting of the system option CONTROLOLDWFL. If CONTROLOLDWFL is SET, the job is prefixed with "BEGIN JOB;"; otherwise, it is prefixed with "JOB;".

All jobs keyed in at an ODT continue to be suffixed with an ";END JOB".

WORK FLOW MANAGEMENT (WFM) SYSTEM ORGANIZATION

This subsection provides a brief introduction to the Work Flow Management (WFM) system. This basic overview of the WFM system also provides an elementary discussion of queue-level scheduling. No attempt is made to give details of the Master Control Program (MCP) procedures and system interconnections mentioned.

Overview

One objective of the WFM operating system is to localize those functions relating to work flow control in order to facilitate the construction of installation supervisor programs. The MCP is organized in order to meet this objective. In particular, all scheduling and operator interface functions are localized in a program unit called the Controller.

The WFM operating system includes four program units: the MCP, the Controller, the WFL compiler, and the Jobformatter. The Controller manages the display routines and the scheduling queues provided by WFM. The WFL compiler handles the control language necessary for job control. Jobformatter provides for the printing of job summary information as an adjunct to printing backup files.

Figure 1-1 shows the interaction among these units. The two external influences on the system are job decks and operator inputs.

WFL Compiler

When a job deck is placed in the card reader, the MCP fires-up the WFL compiler. (The WFL compiler can also be programmatically invoked.) The WFL compiler checks syntax and translates job control statements into machine code. The WFL compiler also creates a special type of file called a jobfile for each job it compiles. Object code to run the tasks of that job, copies of the WFL source (for later printout), data decks belonging to the job, and space for logging and restart information are placed in this file by the WFL compiler. Once compiled, a jobfile is a self-sufficient representation of a job.

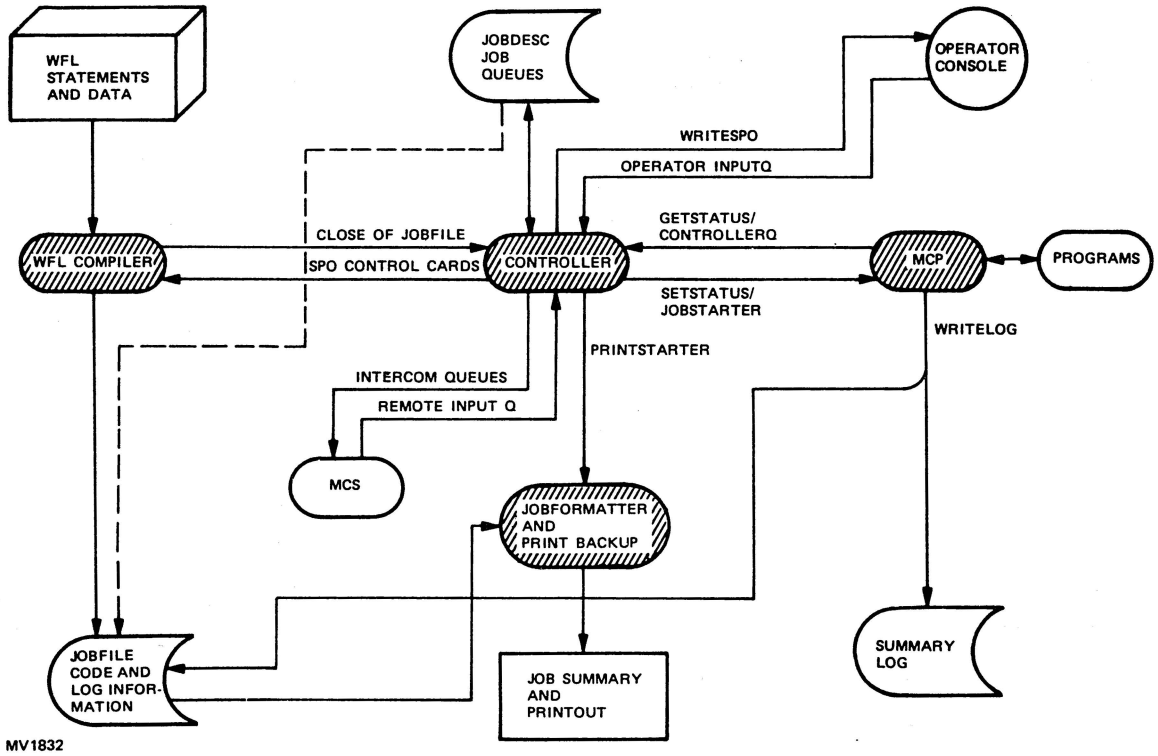


Figure 1-1. WFM System Organization

Controller

Once analysis is completed, the control cards of a job are given to the Controller. The management of the work load is the responsibility of the Controller, which is fired up as an independent runner at Halt/Load time. The Controller is in charge of queue-level scheduling and operator communication. It receives notice of operator inputs and certain other system events thru input queues. The Controller can request, in turn, any useful aspect of the system status and change that status (for example, purge tapes, DS or ST jobs, or MC programs).

Queue-Level Scheduling

Queue-level scheduling is more efficient than MCP level scheduling in two important aspects. First, queue-level scheduling absorbs fewer system resources than MCP-level scheduling, and thus, provides a practical way to make large numbers of jobs visible candidates for system or operator selection. Second, an installation may define multiple queues for different classes of service. For example, the installation can set up one batch queue and one quick-service queue; jobs from both queues are visible when the system starts a new job. Because turnaround of a job in a queue is related to the time required by the longest job in the queue, multiple queues can improve service for short jobs.

The Controller maintains a Jobfile Description File (JOBDESC), which has the dual roles of directory for the jobfile disk areas and queue for the scheduling of the Controller. Entries in JOBDESC consist of the file headers for the jobfiles and links used by the Controller to organize the jobs by class and priority. The Controller procedure ABSTRACT chooses the appropriate class by matching requirements of the job (inserted by WFL in the jobfile) with the specifications of the various queues. The broken line in Figure 1-1 illustrates that the job queues contain pointers to the various jobfiles, ordering them by queue and priority.

The Controller routes jobs to appropriate queues. This procedure may be specified explicitly by a CLASS statement. If a job contains no such statement and has no resource restriction statements, it is put in the default queue, if one exists.

The system provides one queue by default: queue 0. Because the queue number is an implicit priority on the queue, this queue has the lowest priority in scheduling. The installation can modify or delete queue 0 as well as add other queues. However, if no queues exist, no jobs are run. In that case, all jobs are terminated with the error message JOB DS-ED OUT OF QUEUE, which means that no acceptable queue exists for them to join.

When restrictions are specified for a job, the job is put in the queue whose limits it does not exceed. The job is then subjected to two stages of scheduling with different selection rules and resource demands. While the job is in the scheduling queue, selectability depends on the queue number and queue attributes called MIXLIMIT and TURNAROUND. Eventually, the Controller selects the job and presents it to the MCP for processing. At that point, the standard MCP scheduling algorithms based on core estimates and priority take over.

The limits associated with a queue describe the maximum resources that a job from that queue may use. The possible limits are on priority, I/O time, process time, subspaces, tapes allowed (RESOURCE statement), lines printed, and cards punched. For example, if a job has a PROCESSTIME control statement, that statement must indicate less time than the limit of the queue in order to be allowed in that queue. If no limit is specified for the queue, all jobs are acceptable with respect to that attribute. If a job is not acceptable to any queue, it is terminated with a JOB DS-ED OUT OF QUEUE error message. If the job gives no restriction, it can go in any queue; but it is then assigned default restrictions from the queue that it enters. (This procedure is valid only if the compile-time option QFACTMATCHING is set.) See Figure 1-2 for a more complete version of the queue-matching algorithm.

If an attribute is specified as a limit, only jobs that have a lower limit can enter the created queue. If the attribute is specified as a default, any task without an explicit limit on that attribute is assigned the queue default. These defaults are applied at the time the job is selected from the queue to be run, not at the time of insertion in the queue. The limit on PRIORITY applies to operator input for a job in a given queue as well as to jobs being enqueued. The priority of a job may not be placed higher than the limit of the queue in which it is present.

While the job is in the queue, the job can be cancelled, can have the priority changed, or can be explicitly selected. Also, the queues can be interrogated, changed, or can otherwise have attributes manipulated.

If a queue is altered, purged, or eliminated, all jobs waiting to be selected from it are reexamined to determine if they qualify for that queue or, if not, to be redistributed to other queues. However, once a job has been selected and given to the MCP, it is not affected by queue changes unless a Halt/Load occurs. A Halt/Load requeues aborted jobs for restart; therefore, an intervening PQ, MQ, or EQ ODT message may affect those jobs.

Controller-MCP Interface

The Controller procedure SELECTION chooses a job to be started, paying due attention to the queue priorities and mix limits. Using the queue default attributes, this procedure assigns any job attributes that are not explicitly set by the job. The Controller then calls the MCP procedure JOBSTARTER and passes job entry to the procedure. The job is removed from the queue, and JOBSTARTER transfers the job entry to the proper task/stack STRUCTURE to run. JOBSTARTER then enters the jobfile header back in DISKFILEHEADERS and initiates the task thru DOCTOR.

While the job is active (or scheduled at the MCP level), the jobfile is used in the following ways:

1. The jobfile is the code file for job statements; therefore, it is referenced, like any code file, to load executable code in memory.

2. The jobfile contains the card data input needed for the job plus pointers to that data.

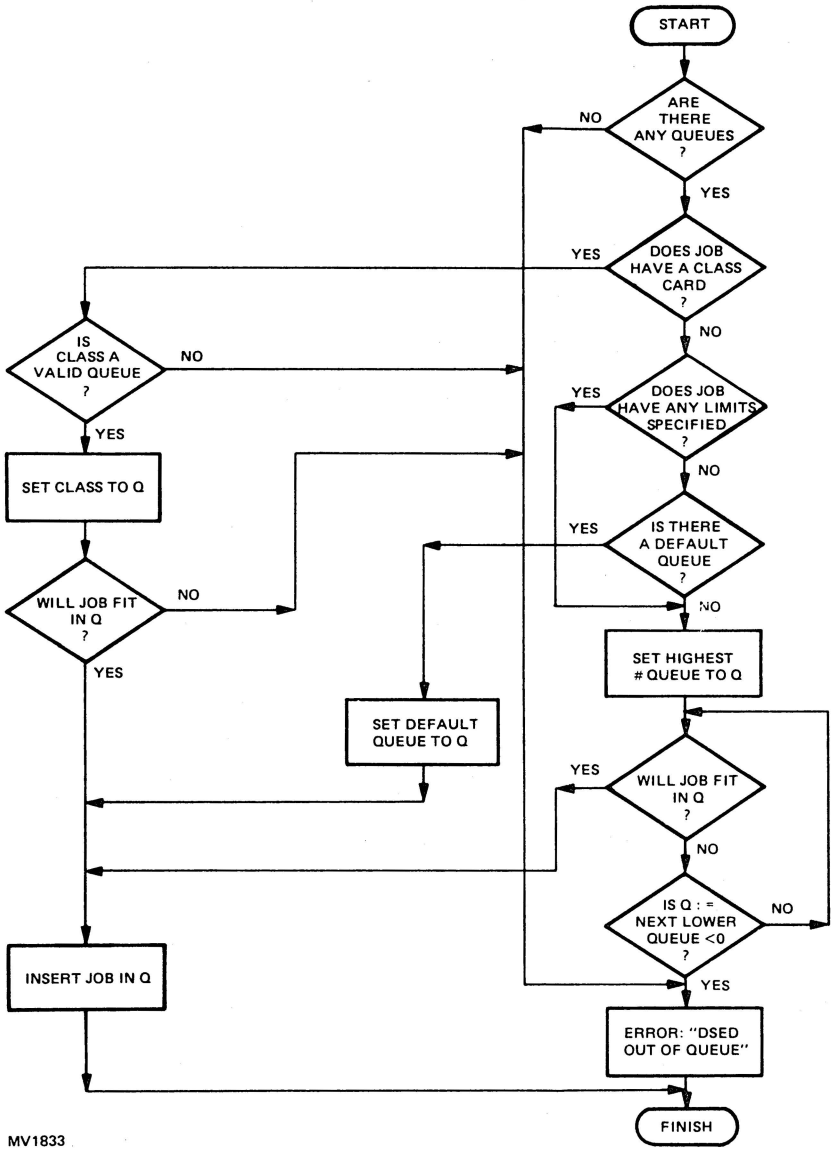


Figure 1-2. Job Enqueueing Algorithm

3. The jobfile is the repository of job-related log entries that, with the control cards, are printed at end-of-job. The logging procedure WRITELOG makes log entries to both the jobfile and the system log (Figure 1-1).

4. The jobfile is used as a roll-out space between complete tasks, so that the job may be properly restarted in the event of a Halt/Load.

At EOJ, the MCP puts an EOJ notice in CONTROLLERQ. The Controller then enqueues the jobfile for printing by adding the jobfile index in the length field of the BD queue word it is building. Thus, jobfiles are queued for printing in the same way as BD files.

After the jobfile is printed, AUTOPRINT continues to print any back-up-printer files for the job. AUTOPRINT also inserts an end-of-printing notice in the CONTROLLERQ, so that the Controller can deallocate the disk space assigned to the jobfile. Some jobs, because of a WFL syntax error or queue limit error, are never run. These jobs are passed directly from the Controller to the AUTOPRINT queue without being run.

In order to communicate with the MCP, the Controller calls two routines, GETSTATUS and SETSTATUS. GETSTATUS returns answers to any interrogations that are made; SETSTATUS performs operator requests, including purging tapes, DSing jobs, and changing the time. All syntax analysis is done in the Controller; the requests given to GETSTATUS and SETSTATUS are in a coded form.

SETSTATUS and GETSTATUS are available to any privileged DCALGOL program. However, the Controller can also get unsolicited information from the MCP by four queues. One queue, the CONTROLLERQ, receives notices about changes of state of the system. This queue includes information used in job scheduling, as well as "events" needed when a terminal has requested event-mode Automatic Display Mode (ADM) message. The second second queue OPERATORINPUTQ. KEYIN places ODT inputs in this queue. The third queue, MESSAGEDISPLAYQ, receives only display and RSVP messages. The fourth queue, REMOTEINPUTQ, receives all MCS input, such as operator input from RJE terminals and various CANDE inputs. The Controller uses another DCALGOL procedure, WRITESPO, to write out to single-line control devices. Because queues and job-level scheduling are managed by the Controller, the Controller is not required to use SETSTATUS or GETSTATUS to respond to queue-related operator inputs but rather handles these inputs without MCP intervention until a specific job must be run or printed.

If a job control deck is typed in at the console (this entry may not include data statements), the Controller processes the WFL compiler to parse the statement and create a jobfile in the normal manner.

The Controller maintains the terminal configuration specifications on disk at the beginning of JOBDESC. That is, if an ADM or TERM message is typed in, the newly effective instructions are saved so that they may be recovered after a Halt/Load.

Logging

The MCP procedure WRITELOG provides the access mechanism to enter fixed-length records. This procedure time stamps them, sends them to the SUMLOG and, if the other parameters and stack status permit, sends them to the jobfile. WRITELOG suppresses logging to either of these destinations, based on the following MCP arrays: SUMLOGOMIT for the system log and JOBLOGOMIT for the jobfiles. When a log request is made, WRITELOG indexes the appropriate array by the log major type and selects a bit, using the log minor type. If that bit is on, logging is suppressed on the respective file. If the major type is greater than the array length or the minor type greater than 47, the entry is logged. By default, open and close records are suppressed from the jobfiles; likewise, aborted history is suppressed from the SUMLOG.

A log entry type (major type 7) is provided for installation use. In order to put records of this type in the log, an installation must appropriate WRITELOG calls to the MCP. (The first four words of these records must still agree in FORMAT with the standard entry types.)

RAILROAD DIAGRAMS

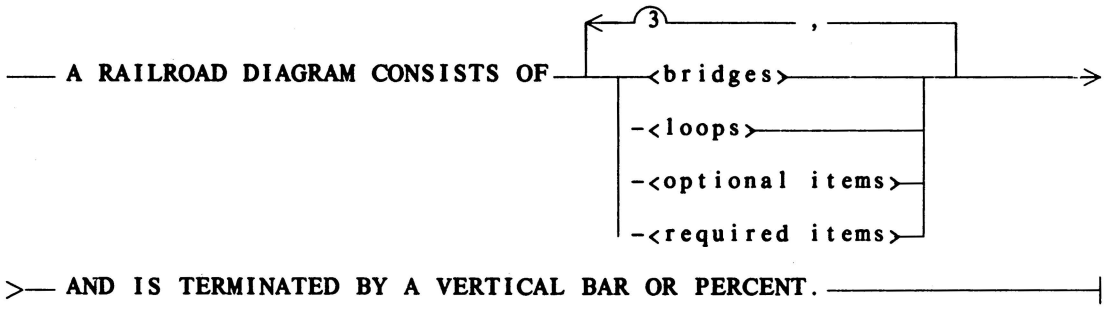
A railroad diagram is a technique used to graphically represent the syntax of language elements. Traversing a railroad diagram from left to right, or in the direction of the arrowheads, and adhering to the limits illustrated by bridges produces a syntactically valid statement. Continuation from one line of a diagram to another is represented by a right arrow (>) appearing at the end of the current line and at the beginning of the next line.

The complete syntax diagram is terminated by a vertical bar (|) or a percent sign (%). Syntax diagrams presented in this document use the convention that a vertical bar (|) terminating a diagram means that the construct may be followed by some other construct on the same card image (as is true in most cases). The percent sign (%) terminating a diagram means that the construct must be the last item on a card image (this requirement applies only to data deck specifications and to the END JOB statement).

Items contained in broken brackets (< >) are syntactic variables which are further defined in the manual or which require requested information to be supplied.

Uppercase items must appear literally. Minimum abbreviations are underlined.

Example



The following statements are examples of syntactically valid statements that can be constructed from the above diagram:

A RAILROAD DIAGRAM CONSISTS OF <bridges> AND IS TERMINATED BY A VERTICAL BAR OR PERCENT.

A RAILROAD DIAGRAM CONSISTS OF <optional items> AND IS TERMINATED BY A VERTICAL BAR OR PERCENT.

A RAILROAD DIAGRAM CONSISTS OF <bridges>, <loops> AND IS TERMINATED BY A VERTICAL BAR OR PERCENT.

A RAILROAD DIAGRAM CONSISTS OF <optional items>, <required items>, <bridges>, <loops> AND IS TERMINATED BY A VERTICAL BAR OR PERCENT.

Railroad Components

<required items>

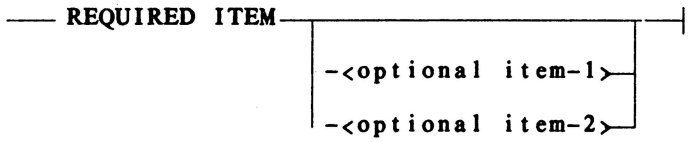
No alternate path through the railroad diagram exists for required items or required punctuation.

Example

<optional items>

Items shown as a vertical list indicate that the user must make a choice of the items specified. An empty path through the list allows the <optional item> to be absent.

Example



The following statements are examples of syntactically valid statements that can be constructed from the above diagram:

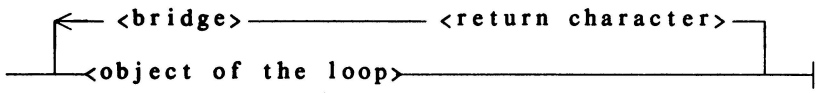
REQUIRED ITEM

REQUIRED ITEM <optional item-1>

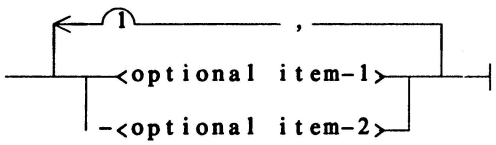
REQUIRED ITEM <optional item-2>

<loops>

A <loop> is a recurrent path through a railroad diagram. A <loop> must be traversed in the direction of the arrowheads, and the limits specified by bridges cannot be exceeded. A <loop> has the following general format:



Example



The following statements are examples of syntactically valid statements that can be constructed from the above diagram:

<optional item-1>

<optional item-1>,<optional item-1>

<optional item-2>,<optional item-1>

<bridges>

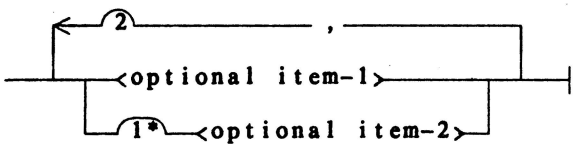
A <bridge> illustrates the minimum or maximum number of times a path may be traversed in a railroad diagram.

Two forms of <bridges> exist:

/ n \ n is an integer that specifies the maximum number of times the path may be traversed.

/ n* \ n is an integer that specifies the minimum number of times the path must be traversed.

Example



The loop may be traversed a maximum of two times; however, the path for <optional item-2> must be traversed at least one time.

The following statements are examples of syntactically valid statements that can be constructed from the above diagram:

<optional item-1>,<optional item-2>

<optional item-2>,<optional item-2>,<optional item-1>

<optional item-2>

2. BASIC ELEMENTS AND CONSTRUCTS

The basic elements and constructs necessary to successfully use WFL are listed and described in this section. Some general facts and restrictions regarding WFL elements are listed below, and many of the basic elements and constructs are syntactically defined on the following pages.

WFL uses the EBCDIC character set; use of any invalid EBCDIC character is illegal except in column one.

Identifiers and numbers are terminated by any nonalphanumeric character (including a blank).

No identifier, constant, string, or multicharacter delimiter may be broken across a card boundary. Numeric constants and multicharacter delimiters may not have embedded blank characters.

No variable, <label id>, or <subroutine id> may be spelled BEGIN, END, JOB, REAL, INTEGER, BOOLEAN, FILE, TASK, STRING, SUBROUTINE, TRUE, FALSE, THEN, ELSE, UNTIL, DATA, EBCDIC, BCL, or BINARY.

The symbol <i> signifies an invalid character punch in column one of a card image (the question mark is usually used for jobs received from non-card-reader sources such as disk files, load control tapes, and ODTs). An <i> is required (1) on the first card of any job or external data deck, (2) on the first card following any data deck, and (3) on the last card of any job. An <i> in column one on any other card of the WFL job is treated as a semicolon. All WFL jobs should begin with an <i> in column one on the first and last statements of the job.

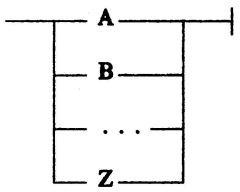
Statements are terminated by a semicolon or by an invalid character <i> in column one on the next card image. Card images may contain more than one statement (properly separated by semicolons). WFL input from the ODT and input from the "ZIP with array" statements of the various programming languages may only present one card image to WFL. In this case, the ? (substituting for the <i>) may only occur as the first character of the input. In other cases where the <i> is required, a semicolon must be used.

Card images may be terminated by a percent sign if such a character is syntactically valid. The remainder of the card image is not scanned and may contain any comments.

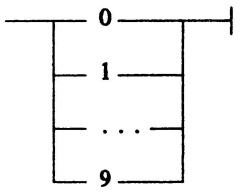
CHARACTER ELEMENTS

The valid character elements are syntactically defined below.

<letter>



<digit>



<string char>

Any character except quote (").

<hyphen>

The single character "-".

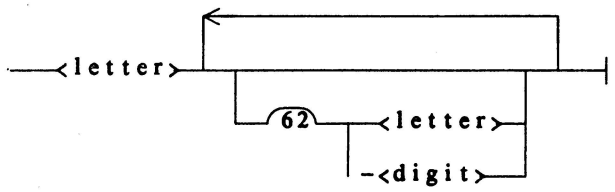
<underscore>

The single character "_".

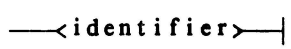
IDENTIFIERS

Valid syntactic diagrams for identifiers are shown below.

<identifier>

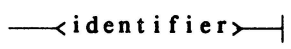


<Boolean id>

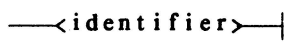


<file id>

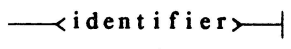
<integer id>



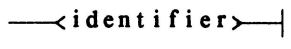
<label id>



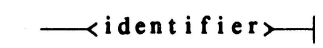
<real id>



<string id>



<subroutine id>



<task id>

Examples

A

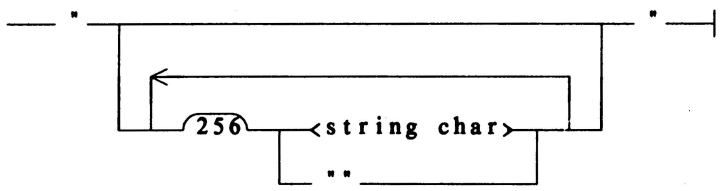
Z123

ABC123

CONSTANTS

String, Boolean, integer, and real constants are syntactically defined below.

<string constant>

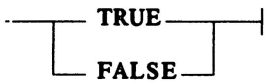


A pair of quote characters ("") appearing alone represents a null string (a string of length zero).

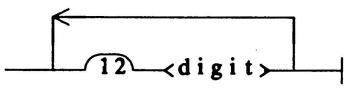
A pair of quote characters ("") appearing in a string represents a single quote character (") within the string.

A <string constant> may not be broken across a card boundary; therefore, the actual maximum length is less than 256 characters.

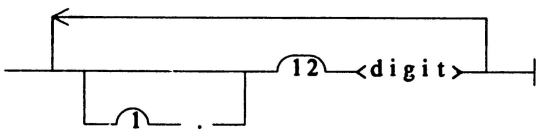
<Boolean constant>



<integer constant>



<real constant>



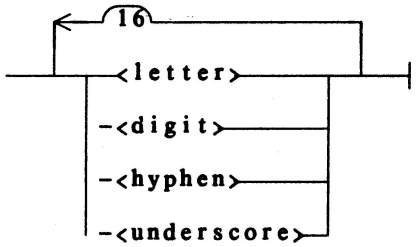
Examples

```
"ABC"           % STRING CONSTANT
"?*->"          % STRING CONSTANT

12              % INTEGER CONSTANT
750             % INTEGER CONSTANT
12345           % INTEGER CONSTANT

3.1416          % REAL CONSTANT
 .2             % REAL CONSTANT
1.0             % REAL CONSTANT
```

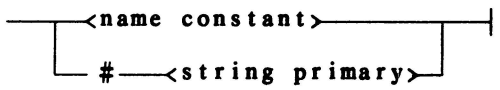
NAMES

Various "name" constructs are syntactically defined below.

<name constant>



<name>



<string primary> is described in Section 4 under **STRING EXPRESSIONS**.

<hostname>

<familyname>

<usercode>

<password>

<intname>

Semantics

When used as a <name>, the #<string primary> must evaluate to the form of a <name constant>; otherwise, a run-time error occurs.

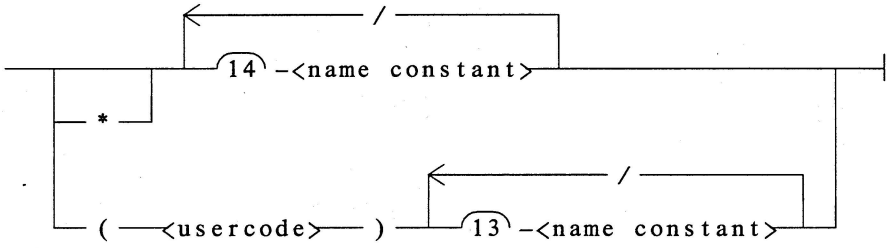
Examples

```
INPUT                          % NAME CONSTANT
FILE06                         % NAME CONSTANT
OUTPUT-FILE                    % NAME CONSTANT

#("NUMBER"&STRING(N,*))        % NAME
MONTH07                        % NAME
```

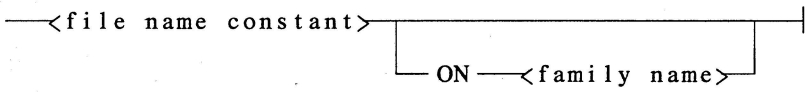
FILE NAMES, TITLES, DIRECTORIES

Syntax

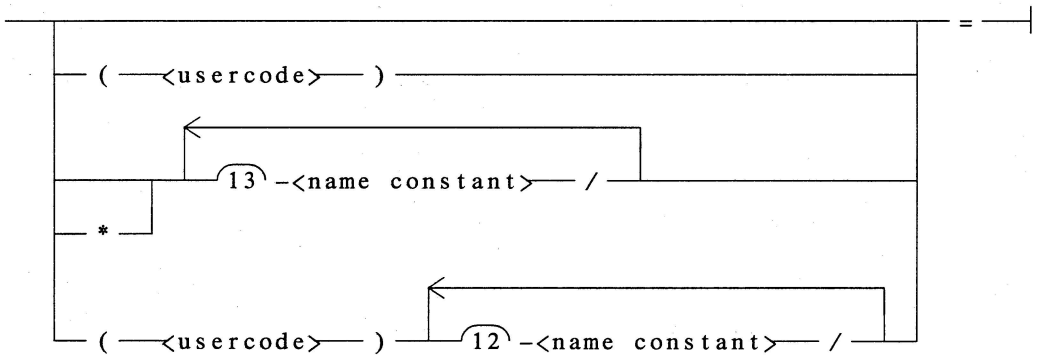
<file name constant>



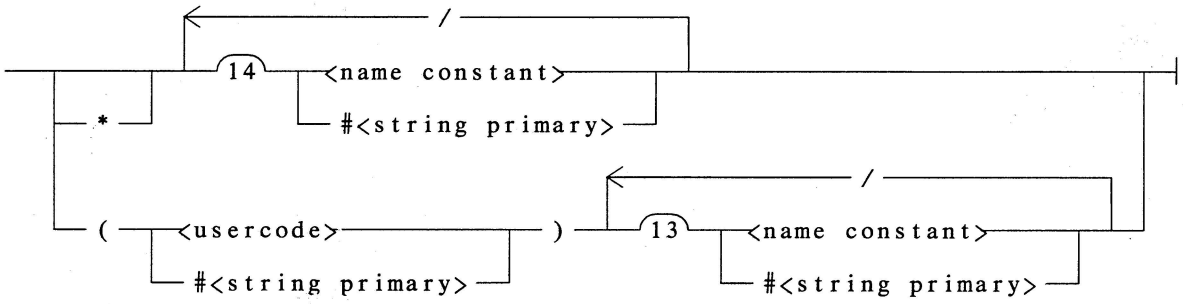
<file title constant>



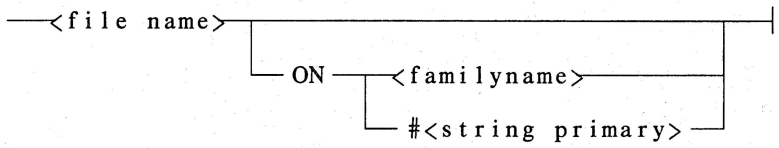
<directory name>



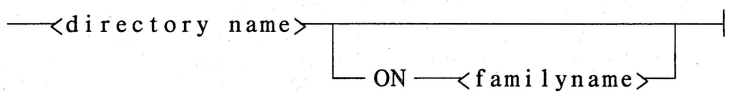
<file name>



<file title>



<directory title>



Semantics

<file name>s and <file title>s may be built dynamically by the use of the "#<string primary>". The run time result must form a valid <file name constant> or <file title constant>; otherwise, a run time error occurs.

Examples

In the following examples, S1 and S2 are <string id>s and F is a <file id>:

```
S1:="B";
F(TITLE = A/#S1/C);              % RESULT TITLE = A/B/C

S1:="(A)B";
F(TITLE = #S1/C);                % RESULT TITLE = (A)B/C
F(TITLE = S1/C);                 % RESULT TITLE = S1/C

S1:="A/B";
S2:="PQ";
F(TITLE = *#(S1 ON S2));         %RESULT TITLE = *A/B ON PQ
```

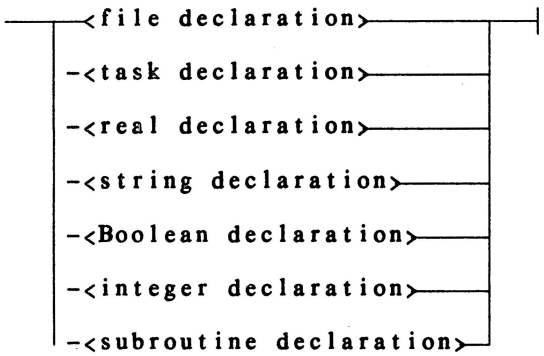
3. DECLARATIONS

Declarations define variable types and express their intended use. As in ALGOL, declarations must precede their corresponding statements.

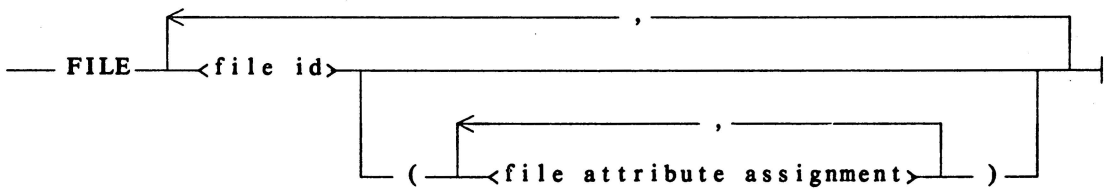
VARIABLE DECLARATIONS

Syntax

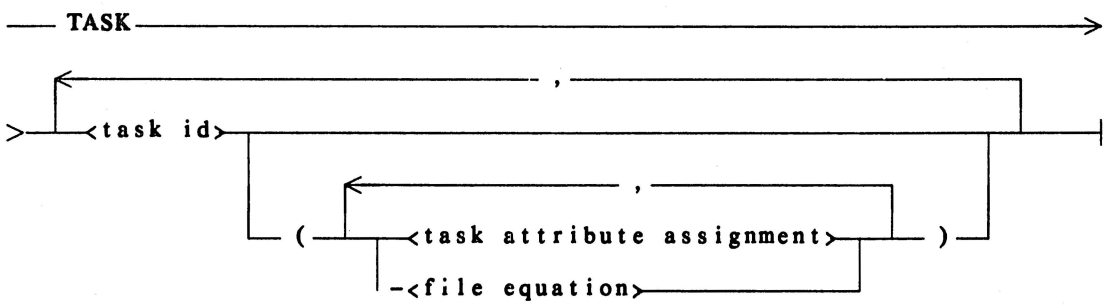
<declaration>



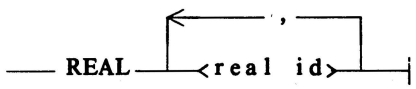
<file declaration>



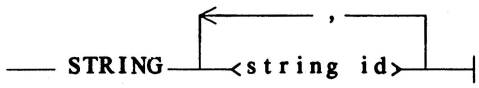
<task declaration>



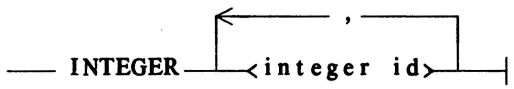
<real declaration>



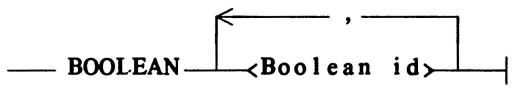
<string declaration>



<integer declaration>



<Boolean declaration>



<subroutine declaration>

The <subroutine declaration> is defined in the following subsection of this section.

Semantics

Declarations may occur either at the job level ("globals") or in WFL subroutines. All declarations in a WFL subroutine specify variables that are local to that subroutine. The ALGOL rules of scope are used for handling global variable references.

File and task attributes in a file or task declaration may only be assigned constant values.

All declarations within a job must follow the <job attribute specification> list and must precede any executable statement in the job. All declarations in a WFL subroutine must follow the BEGIN statement for the subroutine and must precede any executable statement in the subroutine. Declarations may occur in any order. Subroutine declarations may be nested to a depth of ten.

Statement <label id>s are not declared.

All variables used in a WFL job must be explicitly declared.

File declarations may not occur in a subroutine.

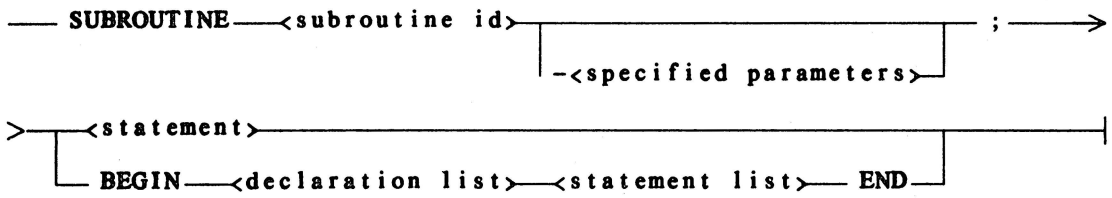
NOTE

The contents of all integer, real, and Boolean variables are saved whenever no tasks are active. When a Halt/Load occurs, the values of all integer, real, and Boolean variables are restored to the most recently saved values. The contents of file, string, and task variables are not saved across a system Halt/Load and are not restored.

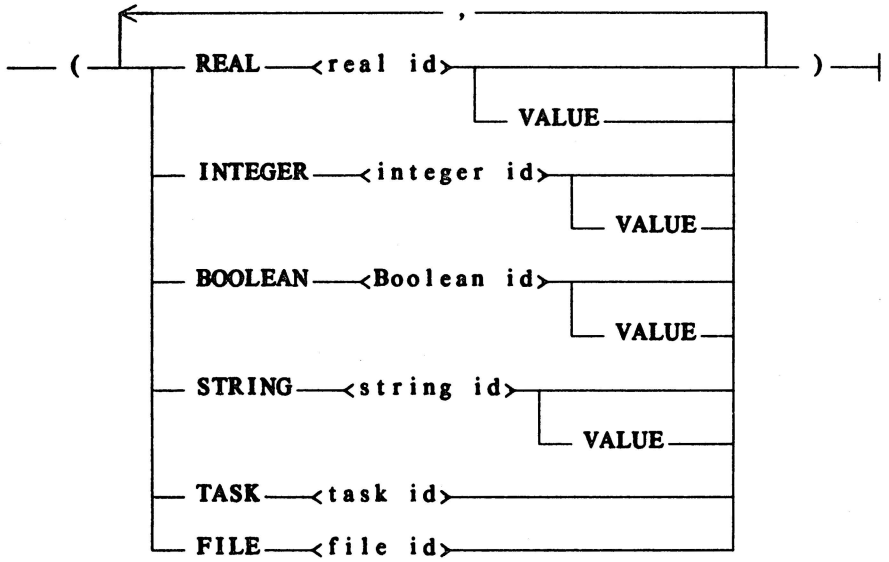
SUBROUTINE DECLARATION

Syntax

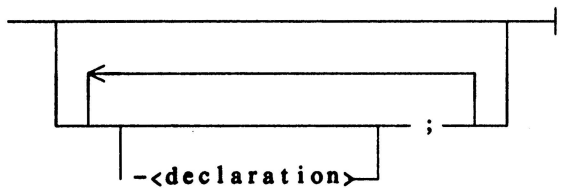
<subroutine declaration>



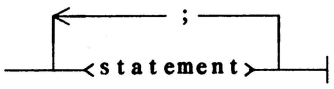
<specified parameters>



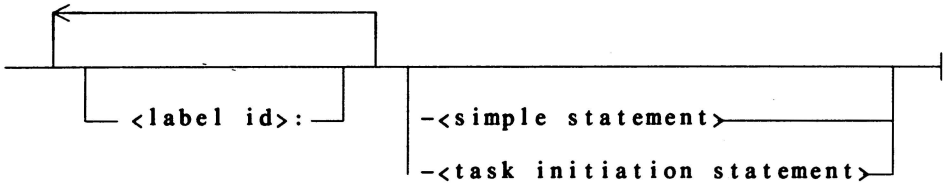
<declaration list>



<statement list>



<statement>



Semantics

The key word preceding the name of a parameter specifies the type of that parameter. The key word VALUE indicates that the parameter is call-by-value rather than call-by-reference.

The same scope rules apply to the names of parameters as to local variables. Declarations in a subroutine may not declare an identifier whose name is the same as any of the parameters of that routine.

Subroutines may be nested a maximum of ten levels. No limit exists on non-nested subroutines. If the current code segment already contains a large amount of code, the WFL compiler generates a new code segment at either the beginning of a subroutine declaration or the beginning of the outer block of the job.

WFL contains inherent <task id>s for MYSELF and MYJOB. These implicit <task id>s exist so that WFL subroutines that are PROCESSed can distinguish between themselves and their parent job task without explicit <task id> declarations.

File declarations may not occur in a subroutine.

The WFL statements are described in Section 6 of this manual.

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

4. EXPRESSIONS

An expression is a meaningful combination of basic elements which, on evaluation, yields a result of a given type (for example, Boolean, real, integer, or string). All variables must be explicitly declared before they are used in an expression.

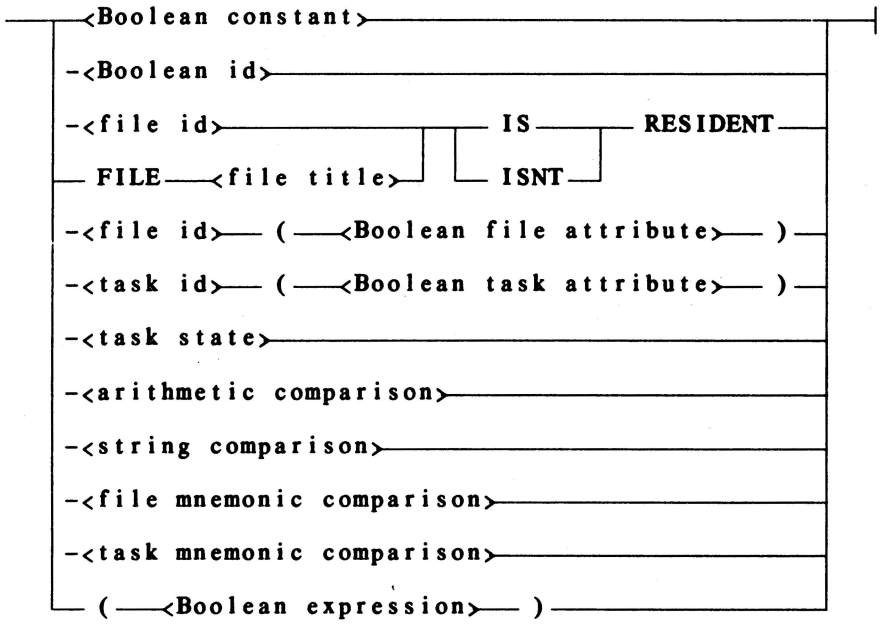
Boolean, real, integer, string, mnemonic, and title <task attribute>s as well as <task mnemonic> are described in Appendix A. Boolean, real, integer, string, mnemonic, and title <file attribute>s as well as <file mnemonic> are described in the B 7000/B 6000 Series I/O Subsystem Reference Manual.

Syntax and semantics for <task attribute assignment>, <file equation>, and <file attribute assignment> are given in the subsections Task Attribute Assignment, File Equation, and File Attribute Assignment in Section 5 of this manual.

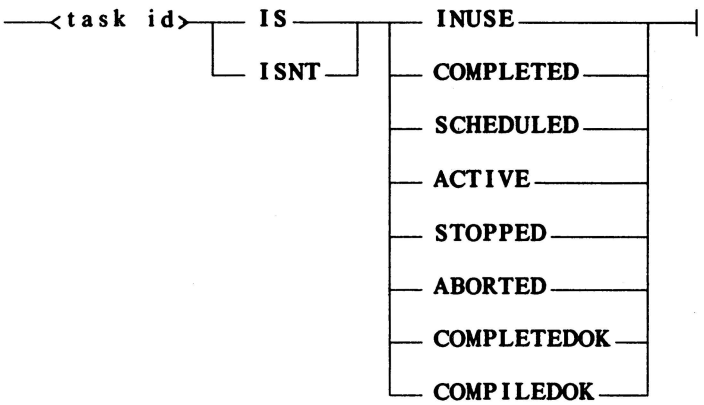
BOOLEAN EXPRESSION

Syntax

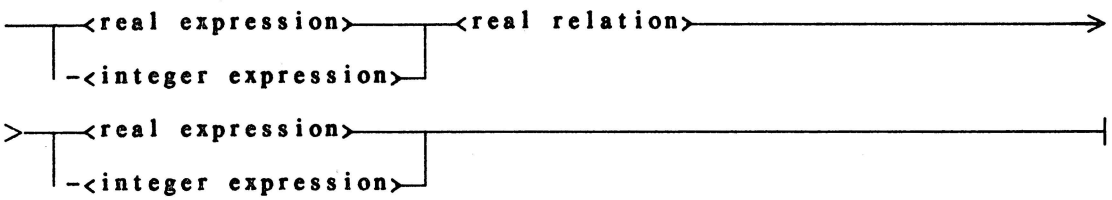
<Boolean primary>



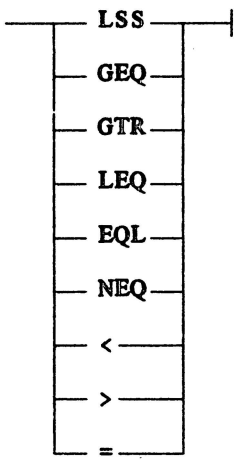
<task state>



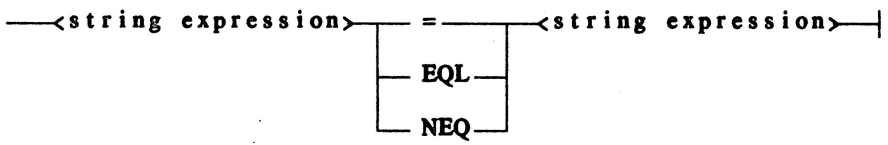
<arithmetic comparison>



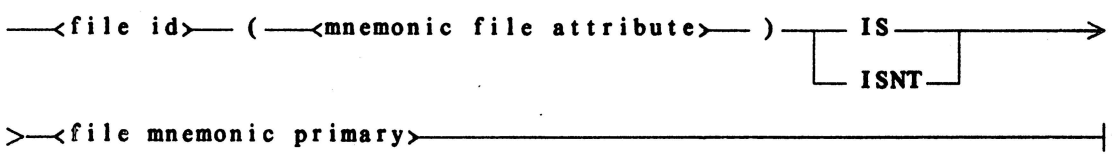
<real relation>



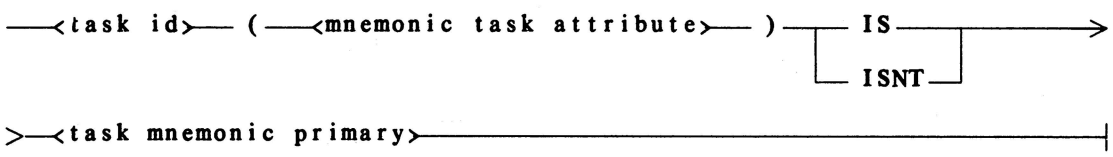
<string comparison>



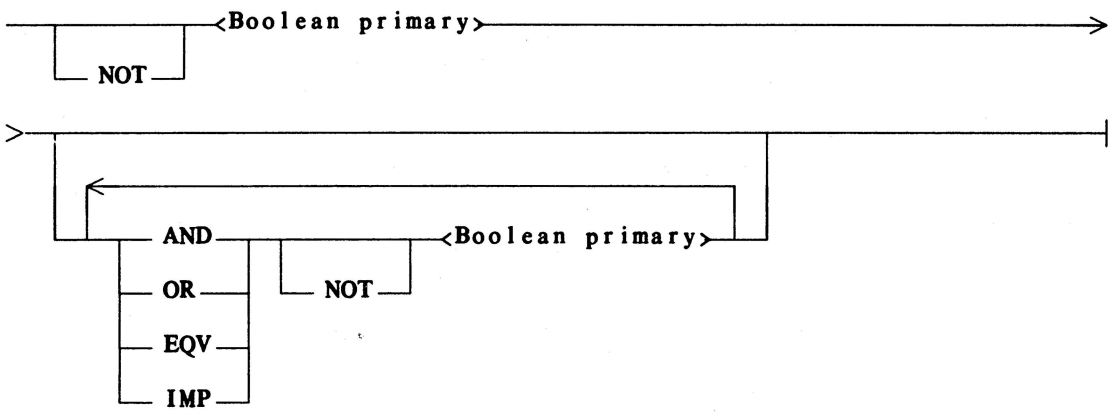
<file mnemonic comparison>



<task mnemonic comparison>



<Boolean expression>



Semantics

Order of Evaluation

The order of priority (highest first) for the execution of Boolean operations is as follows:

1. <Boolean primary>
2. NOT
3. AND
4. OR
5. IMP
6. EQV

All <Boolean primary>s are evaluated first; then the NOT operators are applied to the <Boolean primary>s that they precede. The other operations are performed in decreasing order of priority. If two operations are of the same priority, the left operation is performed first. If a <Boolean expression> is enclosed in parentheses, it becomes a <Boolean primary>.

String Comparison

<string comparison> is a <Boolean primary> that allows comparison of the values of two <string expression>s. Two strings are equal only if all characters in the first string occur in the same order in the second string and the lengths of the two strings are equal.

Task State

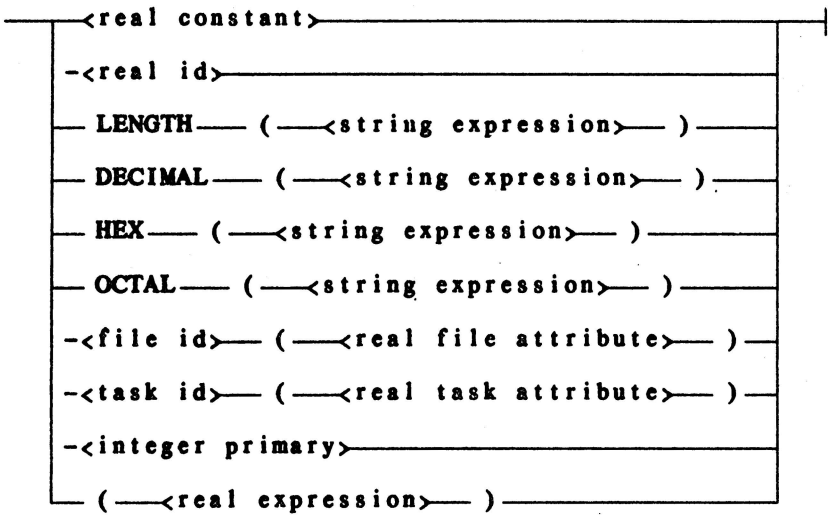
<task state> allows the status of a task to be monitored. The task must be associated with a <task identifier>.

INUSE	The task is SCHEDULED, ACTIVE, or STOPPED.
COMPLETED	The task was initiated and has terminated.
SCHEDULED	The task has not yet been initiated by the system.
ACTIVE	The task is currently running.
STOPPED	The task was stopped by the operator, suspended by the system, or programmatically suspended.
ABORTED	The task faulted or was DSed.
COMPLETEDOK	The task is completed and was terminated without faulting or being DSed.
COMPILEDOK	The task compiled without syntax errors.

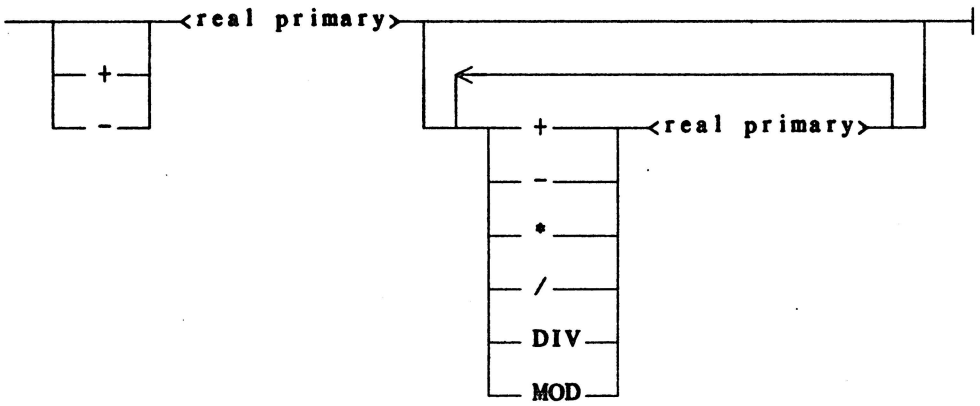
REAL AND INTEGER EXPRESSIONS

Syntax

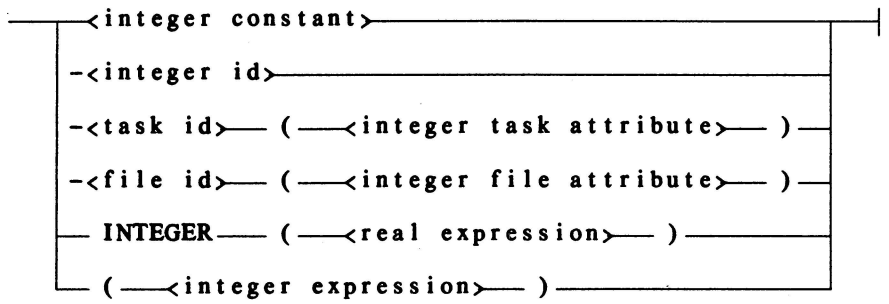
<real primary>



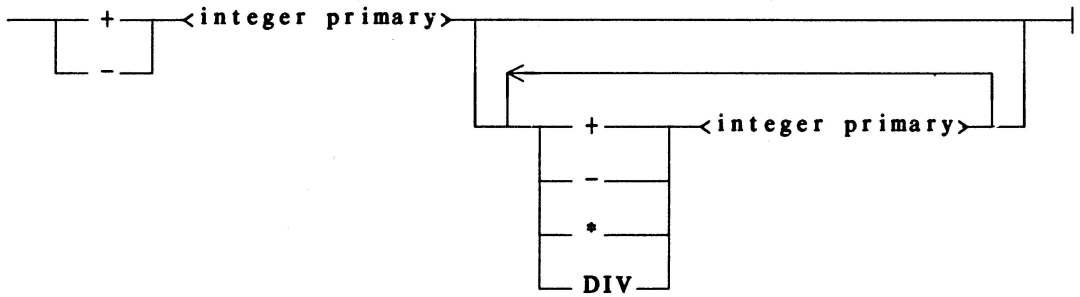
<real expression>



<integer primary>



<integer expression>



Semantics

The order of priority (highest first) for arithmetic operations is as follows:

1. <real primary> or <integer primary>
2. prefix + or -
3. *, /, DIV, or MOD
4. infix + or -

First, all <primary>s are evaluated; second, the prefix + or - (if any) is applied to the <primary> that it precedes. Finally, the remaining operations are performed in decreasing order of priority. If two operations are of the same priority, the left operation is performed first. When an <expression> is enclosed in parentheses, it becomes a <primary>.

The DIV operator produces a quotient with a truncated fractional part; the slash operator produces a quotient that preserves the fractional part.

LENGTH Function

The function LENGTH(<string expression>) is a <real primary> that returns the number of characters in the value of <string expression>.

OCTAL Function

The function OCTAL <string expression> is a <real primary> that returns a real value equal to the octal (base eight) number represented by the value of <string expression>. <string expression> must contain at least one and not more than 16 characters. All characters in the value of <string expression> must be within the set of characters "01234567". A run-time error is given if <string expression> does not satisfy these requirements.

HEX Function

The function HEX(<string expression>) is a <real primary> that returns a real value equal to the hexadecimal (base 16) number represented by the value of <string expression>. <string expression> must contain at least one and not more than 12 characters. All characters in the value of <string expression> must be within the set of characters "0123456789ABCDEF". A run-time error is given if <string expression> does not satisfy these requirements.

DECIMAL Function

The function DECIMAL(<string expression>) is a <real primary> that returns a real value equal to the decimal (base 10) number represented by the value of <string expression>. <string expression> must contain at least one and not more than 12 characters. All characters within the value of <string expression> must be within the set of characters "0123456789". A run-time error is given if <string expression> does not satisfy these requirements.

INTEGER Function

The function INTEGER (<real expression>) is an <integer primary> with a result equal to the <real expression> but without a fractional part.

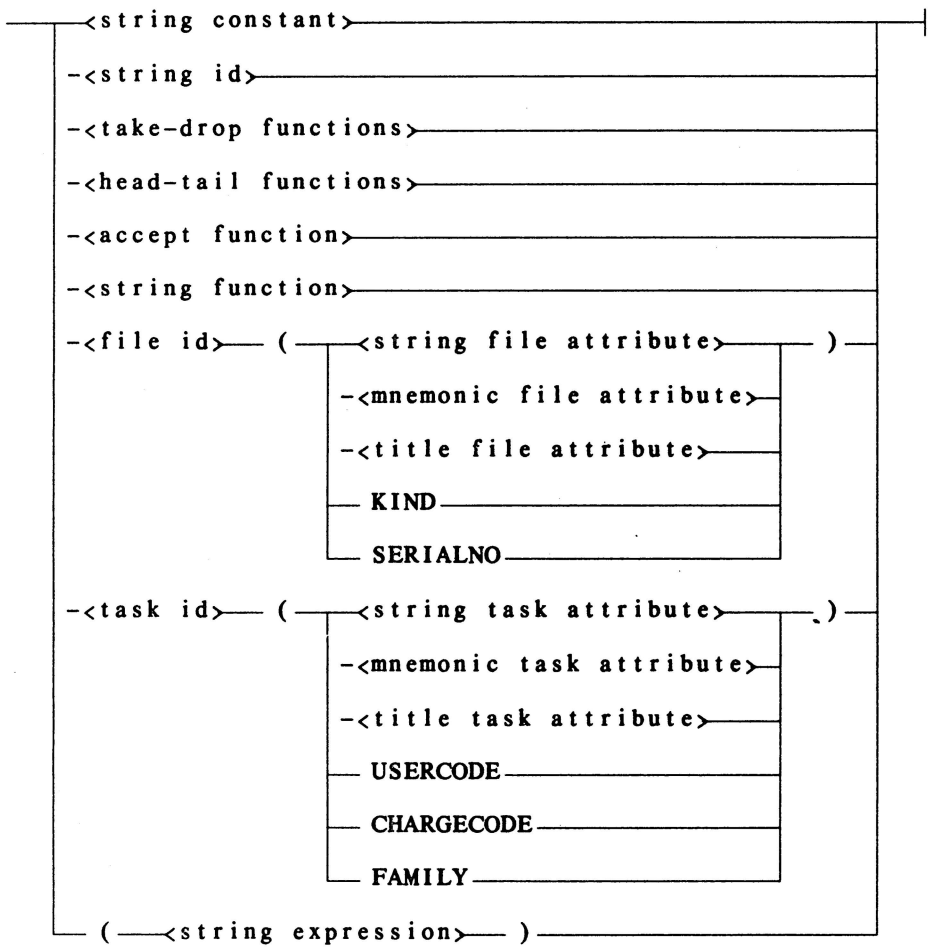
Examples

```
INTEGER(123.456)        % YIELDS 123

LENGTH("ABCDEF")        % YIELDS 6

OCTAL("10")             % YIELDS 8 (DECIMAL)
OCTAL("377")            % YIELDS 255 (DECIMAL)

HEX("10")               % YIELDS 16 (DECIMAL)
HEX("FF")               % YIELDS 255 (DECIMAL)

DECIMAL("10")           % YIELDS 10 (DECIMAL)
DECIMAL("255")          % YIELDS 255 (DECIMAL)
```

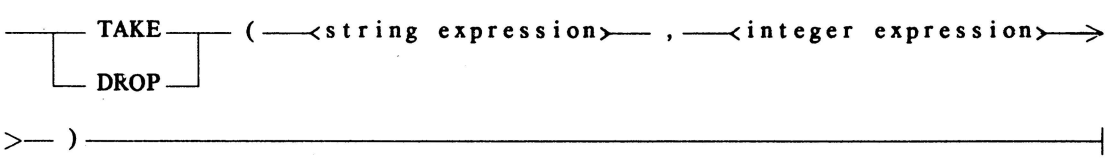
STRING EXPRESSIONS

Syntax

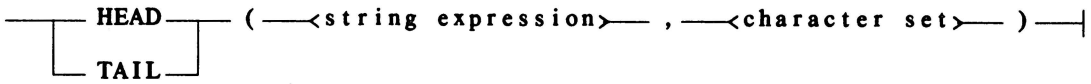
<string primary>



<take-drop functions>



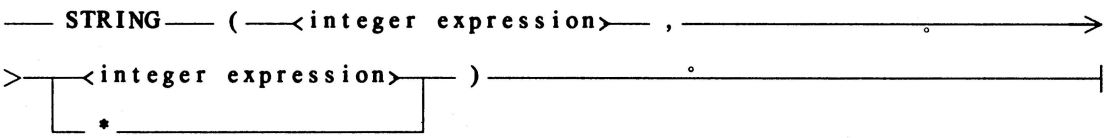
<head-tail functions>



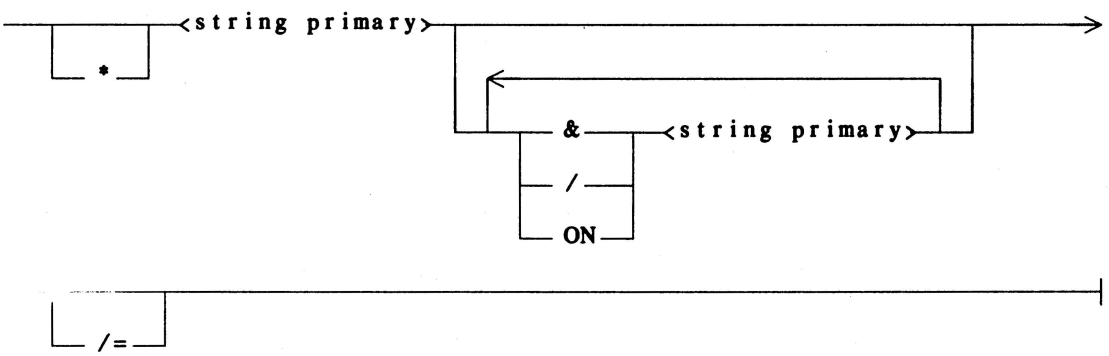
<character set>

<accept function>

<string function>



<string expression>



Semantics

Concatenation Operation

Two or more strings may be concatenated by use of the & operator. The concatenation of two strings yields a new string whose length is the sum of the lengths of the two original strings. The new string is formed by joining a copy of the second string to the end of a copy of the first string; the value of new string is the sum of the values of the two original strings. No string that contains more than 256 characters may be formed; a run-time error results for an attempt to do so.

NOTE

For the 2.9, 3.0, 3.1, and 3.2 system releases, the ability to use the file attribute SERIALNO as a string primary is not implemented.

Other Concatenation Operations

Four other string concatenation operations are provided. These operators are intended to aid building <file title>s from strings; however, the operators need not be used exclusively for that purpose. These string operators may be used to form invalid file titles; a run-time error then results in the job. These operators are as follows:

*	unary prefix asterisk
/	binary infix slash
ON	binary infix ON
/=	unary postfix slash-equal

The action of these operators is described in terms of the string concatenation operator (&); s1 and s2 below represent any possible <string primary>s.

```
*s1         is equivalent to    "*" & s1
s1/s2       is equivalent to    s1 & "/" & s2
s1 ON s2    is equivalent to    s1 & " ON " & s2
s1/=        is equivalent to    s1 & "/="
```

All string concatenation operations are performed from left to right.

A maximum of 1024 strings may be concatenated.

TAKE and DROP Functions

The TAKE function returns a new string whose value is a copy of the first <integer expression> number of characters taken from the <string expression>. If the value of <integer expression> is greater than the number of characters in <string expression> or less than zero, then a run-time error occurs.

The DROP function returns a new string whose value is a copy of the characters remaining in <string expression> after the first <integer expression> number of characters have been discarded. If the value of <integer expression> is greater than the number of characters in <string expression> or less than zero, then a run-time error occurs.

For any <string expression> S and any <integer expression> I in the range 0 LEQ I LEQ LENGTH(S), the following relation is always true:

```
S = TAKE(S,I) & DROP(S,I)
```

HEAD and TAIL Functions

A <character set> is a collection of characters used to control the action of the HEAD and TAIL functions. The inherent <character set> ALPHA consists of the letters A thru Z and the digits 0 thru 9. The <string constant> may have characters in any order. NOT is used to indicate a character set consisting of all EBCDIC characters except those specified in the NOT statement.

The HEAD function returns a new string consisting of a copy of all leading characters in <string expression> which belong to the set of characters specified by <character set>. If the first character in <string expression> is not a member of the <character set>, a null (zero length) string is returned.

The TAIL function returns a new string consisting of a copy of all the characters in <string expression> which remain after the removal of all the leading characters belonging to <character set>. If all characters in <string expression> are members of the specified <character set>, a null string is returned.

For any <string expression> S and any <character set> C, the following relation is always true:

```
S = HEAD(S,C) & TAIL(S,C)
```

ACCEPT Function

The ACCEPT function displays the <string expression> parameter on the ODT and waits for an operator response (via the AX ODT message). The first 256 characters of the operator response are returned as the value of the ACCEPT function. The CONTROLLER (in the MCP) removes all screen control characters found in the <string expression> parameter before it is displayed on the ODT.

STRING Function

The STRING function generates a new string whose value is the value of the decimal representation of the absolute value of the first <integer expression>. The length of the returned string is specified by the second parameter; if this second parameter is an <integer expression> with a value less than or equal to zero, the returned string is of length zero. If the value of the second <integer expression> is greater than the minimum number of characters needed to represent the first argument, a sufficient number of leading zeros are provided. If the value of the second <integer expression> is less than the number of characters needed to represent the first <integer expression>, the right-most characters are returned. If the second parameter of the STRING function is an asterisk, the string is long enough to contain all the digits of the integer character representation of the first argument with no leading zeros.

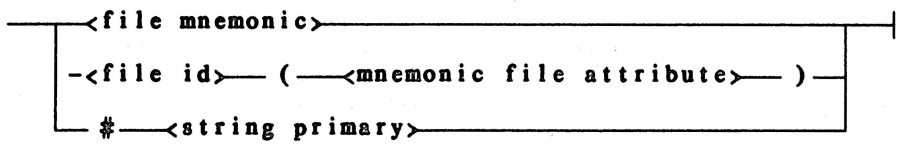
Examples

```
STRING STR1, STR2, PGMNAME;

STR1 := "ABCDEF";
STR2 := TAKE(STR1,2);                  % RESULT = "AB"
STR2 := DROP(STR1,2);                  % RESULT = "CDEF"

STR1 := "     A B C ";
STR2 := HEAD(STR1," ");                % RESULT = "     "
STR2 := TAIL(STR1," ");                % RESULT = "A B C "

STR1 := "FILE/NAME";
STR2 := HEAD(STR1,NOT "/");            % RESULT = "FILE"
STR2 := TAIL(STR1,NOT "/");            % RESULT = "/NAME"

STR1 := STRING(123,*);                 % RESULT = "123"
STR2 := STRING(123,6);                 % RESULT = "000123"
STR1 := STRING(123.456, 7);            % RESULT = "0000123"
STR2 := STRING(1234,3);                % RESULT = "234"

STR1 := "X/";
STR2 := "Y";
PGMNAME := STR1 & STR2 & " ON PAK";    % RESULT = "X/Y ON PAK"

STR1 := "A";
STR2 := "B";
PGMNAME := STR1/STR2/= ON "ABC";       % RESULT = "A/B/= ON ABC"
```

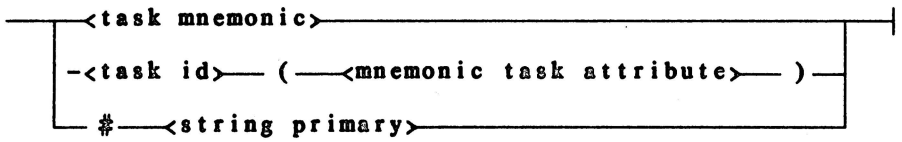
MNEMONICS

Syntax

<file mnemonic primary>



<task mnemonic primary>



Semantics

<file mnemonic primary> and <task mnemonic primary> allow mnemonic-valued attributes to be used effectively in comparisons and assignments. In mnemonic attribute comparisons and assignments, the specific attributes and mnemonics must be compatible. For example, the MYUSE attribute of one file (or one of the MYUSE mnemonics CLOSED, IN, OUT, or IO) may be compared with or assigned to the MYUSE attribute of some other file identifier. The MYUSE and KIND file attributes are not compatible and may not be used together in the same comparison or assignment. The value of the <string primary> must also represent a compatible mnemonic.

<mnemonic task attribute> and <task mnemonic> are described in Appendix A of this manual. <mnemonic file attribute> and <file mnemonic> are described in the B 7000/B 6000 Series I/O Subsystem Reference Manual.

Pragmatics

The restrictions on string variables are as follows:

1. As with file variables and task variables in previous versions of WFL, the contents of string variables are not saved across a system Halt/Load. An ON RESTART statement may be necessary to ensure that the job resets the values of these variables.

2. The form of <name> (in a <file title>) that includes the #S <string primary> may not be used in the LOG or PB statements nor may it be used within any of the library maintenance statements (COPY, ADD, REMOVE, CHANGE, SECURITY, CATALOG, and VOLUME).

Examples

In the following examples, F is a <file id>, T is a <task id>, and S is a <string id>.

```
S:="ALGOL";
IF F(FILEKIND) IS #(S&"SYMBOL") THEN ...

S:="NEVERUSED";
T(STATUS=#S);

STRING DISK;
DISK:="PRINTER";
F(KIND=#DISK);              % YIELDS KIND = PRINTER
```

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

5. JOB STRUCTURE

INTRODUCTION

A program written in WFL describes a job. During execution, a job executes from its own job stack. A task is any process initiated by the job. Examples of tasks are compilations, user program executions, and library maintenance tasks. The job stack controls the execution of its own tasks.

This section is divided into the following subsections:

JOB SYNTAX AND STRUCTURE
JOB ATTRIBUTE SPECIFICATION
TASK INITIATION
- Task Equation
- Task Attribute Assignment
- File and Database Equation
- File Attribute Assignment
- Data Specification

Although most jobs execute from their own job stack, some jobs execute interpretively.

If a WFL job is to be executed interpretively, it must meet the following two conditions:

1. The job must originate either from the CONTROLLER or from a user program by way of a ZIP statement; for example, in ALGOL: "ZIP WITH <array>".

2. The job must consist of a single REMOVE, COPY, CHANGE, SECURITY, START, or RERUN statement.

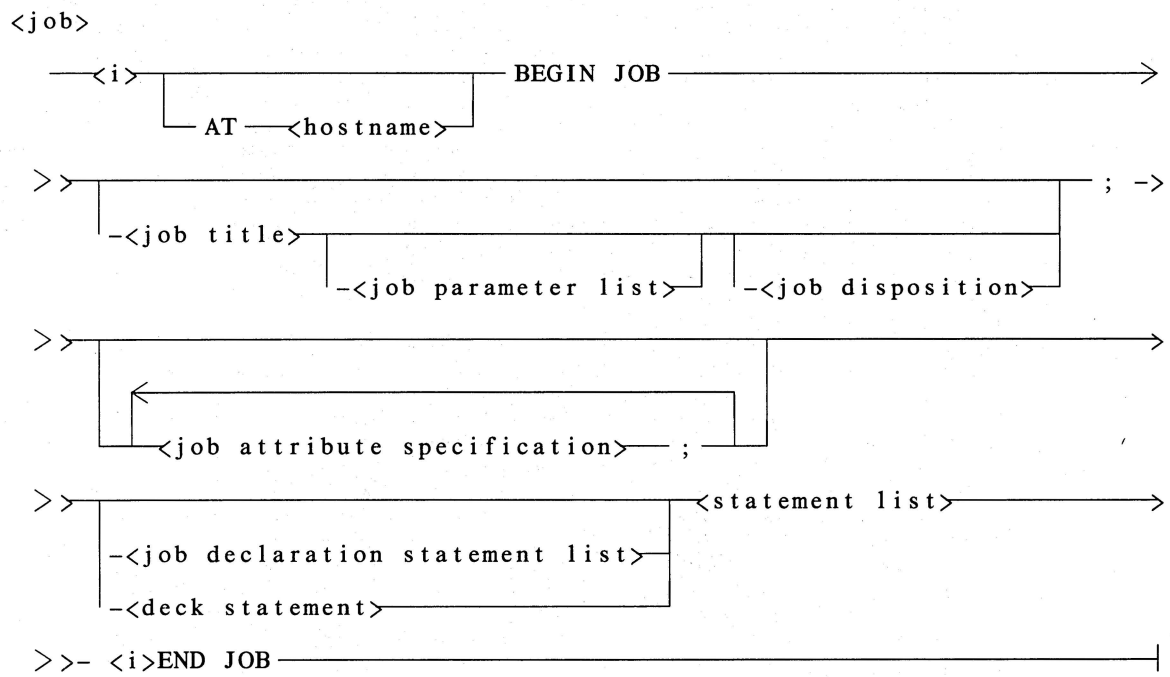
Under these conditions, WFL executes the statement interpretively; that is, a codefile is not generated, and the statement is executed out of the WFL stack.

A WFL job that is started from the ODT and does not contain a USERCODE specification runs without a usercode and, in general, is not privileged. However, WFL jobs that are executed interpretively and do not run under a usercode are privileged; for example, a REMOVE statement that is entered at the ODT would be privileged.

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES

JOB SYNTAX AND STRUCTURE

This subsection presents the syntax for a complete job. The individual syntactic elements are defined on the following pages, and, finally, an example of a job is given.



WFL jobs may be sent from one host to another using an AT <hostname> between the invalid character and the BEGIN JOB on the first card. In the following example, the job is sent to HOSTC:

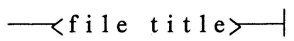
```
<I>AT HOSTC BEGIN JOB X;
    .
    .
    .
<I>END JOB
```

The WFL compiler at the receiving host system compiles the job; the sending host does not analyze the contents of the job (except for the ?END JOB).

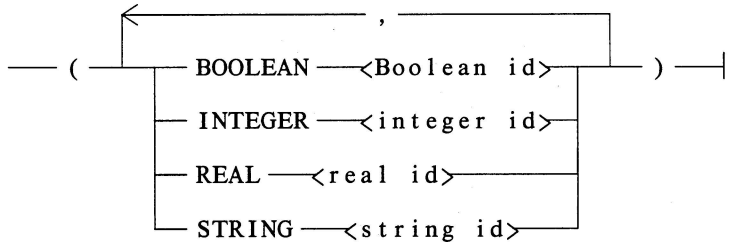
If a job is to be sent to a remote host (?AT <hostname> before BEGIN JOB), the job must not contain BCL or BINARY data decks. Any occurrence of a BCL or BINARY deck causes the following syntax error:

```
BCL OR BINARY DECK NOT ALLOWED IN REMOTE JOB TRANSFER
```

<job title>



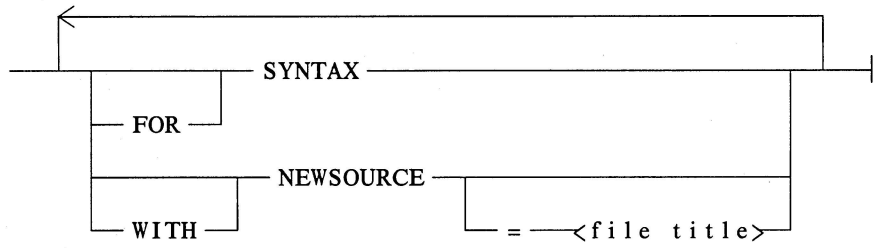
<job parameter list>



A <job parameter list> declares named constants (not variables) of the type specified in the parameter list. A named constant of the appropriate type may be used in any context in which <Boolean constant>, <integer constant>, <real constant>, or <string constant> is allowed.

A WFL job with parameters presented to the system from any source that does not allow the actual parameters to be given (such as SITE or RJE readers, system ODTs, or ZIP statements of the various other languages) must be compiled for SYNTAX. If the job is also compiled with the NEWSOURCE option, a START statement in a subsequent job from any source (including the START command in CANDE) can pass the proper parameters and start the job.

<job disposition>



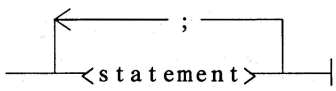
The name of the job is (1) specified by the <file title> in the BEGIN JOB statement; (2) specified by the NAME task attribute in the <job attribute specification> list; or (3) devised by the WFL compiler when no explicit NAME attribute is given.

The NEWSOURCE specification may not occur in jobs received from "ZIP WITH <array>" statements from the various programming languages, from the CANDE WFL command, or from the ODT. The NEWSOURCE specification is ignored for all jobs that are received from disk sources (START or "ZIP WITH <file>" statements).

The WFL compiler assumes that if a job residing on disk contains a NEWSOURCE specification, the job is executable. WFL therefore ignores any syntax specification that may be present.

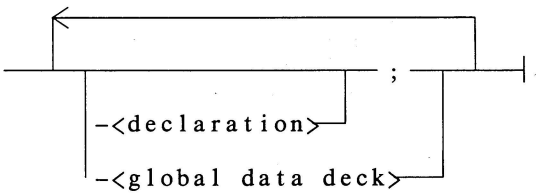
The <job attribute specification>s are defined in a subsequent subsection of this section.

<statement list>

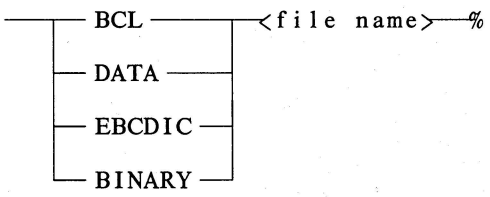


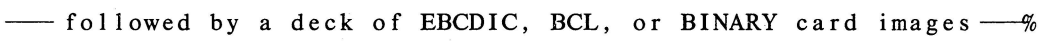
The <statement list> allows the user to supply a series of <statement>s to define the tasks to be performed by the job. The <statement>s are described in Section 6.

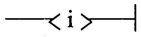
<job declaration statement list>



<global data deck>







A <global data deck> may not occur in a WFL subroutine. A BINARY deck must be terminated by a binary END (BEND) card.

DIAGRAM DELETED

The <deck statement> is discussed in Section 6 under DECK Statement.

Example

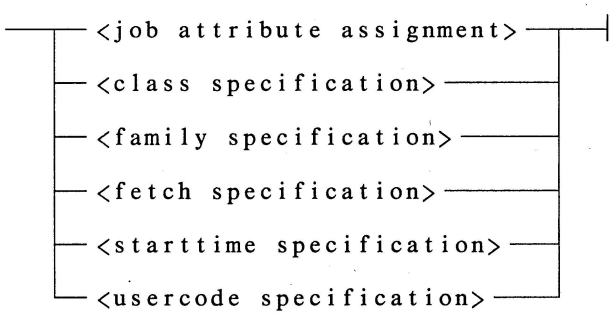
The following is an example of a full job.

```
<i>BEGIN JOB EXAMPLE (STRING TESTNAME, INTEGER TESTNUMBER);
   NAME=EXAMPLE/#STRING(TESTNUMBER,*);
   USERCODE = WFL/MANUAL;
   CLASS = 2;
   TASK TCOMP,TRUN;
   COMPILE #(TESTNAME & STRING(TESTNUMBER,1))[TRUN] PL/I[TCOMP];
   PL/I DATA
        P: PROC;
             DCL I INIT(15);
             DISPLAY ('P IS RUNNING');
             DISPLAY ('NOW ABORT');
             I=I/0;
        END P;
<i>IF TCOMP IS COMPILEDOK THEN
     DISPLAY "COMPILED OK"
   ELSE ABORT "*DID  NOT COMPILE";
   IF TRUN IS COMPLETEDOK THEN
     DISPLAY "RAN OK"
   ELSE ABORT "* RUN ABORTED";
<i>END JOB
```

JOB ATTRIBUTE SPECIFICATION

Syntax

<job attribute specification>



Semantics

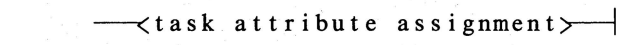
File attribute equation in the <job attribute specification> list is not allowed. The <job attribute specification> list is terminated by the appearance of any statement that is not a <job attribute specification>. A statement beginning with FILE is interpreted as a <file declaration> (part of <job declaration statement list>) and, thus, terminates the <job attribute specification> list.

The following pages contain detailed descriptions of the specific job attribute specifications.

Job Attribute Assignment

Syntax

<job attribute assignment>



Semantics

The <job attribute assignment> is used to assign task attributes to the job. Refer to TASK INITIATION in this section for a description of <task attribute assignment>.

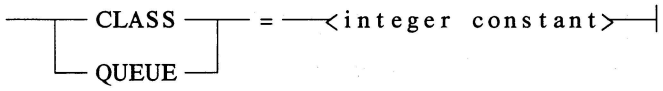
Examples

```
STACKLIMIT   = 2560

PRINTLIMIT   = 10000

MAXPROCTIME  = 60

CHARGECODE   = ABC
```

Class Specification

Syntax

<class specification>



Semantics

The <class specification> is used to assign the CLASS task attribute (the number of the queue desired) for the job.

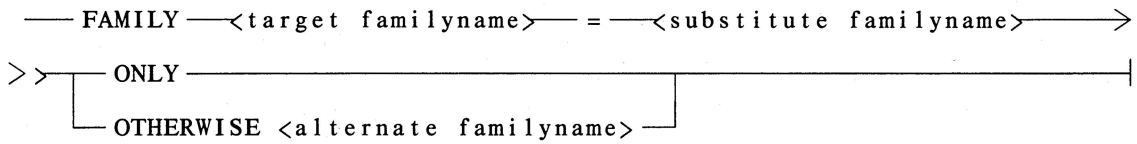
Examples

```
CLASS = 5

CLASS = 60
```

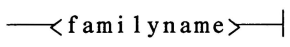
Family Specification

Syntax

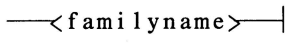
<family specification>



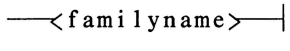
<target familyname>



<substitute familyname>



<alternate familyname>



Semantics

The <family specification> syntax (family substitution) is used to equate a target familyname with a substitute familyname (ONLY) or to equate a target familyname with a substitute familyname and an alternate familyname (OTHERWISE).

When family substitution is in effect, any reference to the target familyname becomes a reference to the substitute familyname. The effect of an alternate familyname varies with the context as follows:

1. If a file cannot be found on a <substitute familyname>, the <alternate familyname> is searched.

2. When a file is being created, the <alternate familyname> is ignored.

3. A CHANGE or REMOVE function through a job affects both <substitute familyname>s and <alternate familyname>s.

<family specification>s are applied to a specific job and are inherited by any tasks run from that job. A job (or task) may have only one family substitution in effect. As with other <job attribute specification>s, the occurrence of a second <family specification> is syntactically correct; however, it overrides the first occurrence.

Examples

In the first example below, all files created are on family DISK. Files are sought first on DISK, then on MYPACK. In the second example, all files created are on MYPACK, and files are sought first on MYPACK, then on DISK. In the third example, disk files are created and sought on system-resource pack only. All disk files that are sought, including compilers and utilities, must be present on PACK. In the fourth example, references to family THISPACK become references to family THATPACK. References to family DISK are unaffected.

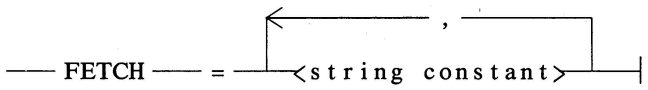
```
FAMILY DISK = DISK OTHERWISE MYPACK

FAMILY DISK = MYPACK OTHERWISE DISK

FAMILY DISK = PACK ONLY

FAMILY THISPACK = THATPACK ONLY
```

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

Fetch Specification

Syntax

<fetch specification>



Semantics

The <fetch specification> is used to display a string on the ODT before a job is run. This specification allows the operator to see which resources are required to run the job; that is, information is displayed which indicates that a job requires operator action. The instructions are displayed by entering the PF ODT message. The job initiation is inhibited until an OK or DS message is entered in response or until the system option NOFETCH is set.

When a job with a FETCH is waiting in a queue, an ODT PF message is used to display the message.

System option 19 (NOFETCH) must be reset.

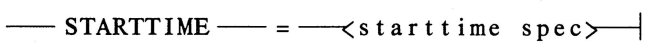
Examples

```
FETCH = "MYPACK IS 3RD VOL", "PACK/A/B/C IS 4TH"

FETCH = "MOUNT TAPED/LIB/WHEN"
```

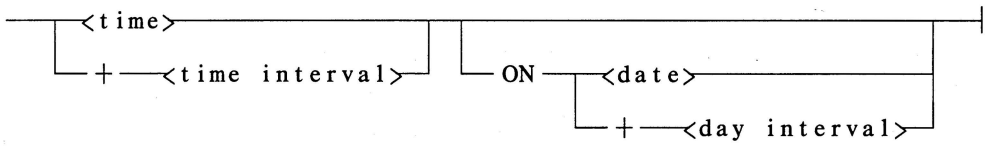
Starttime Specification

Syntax

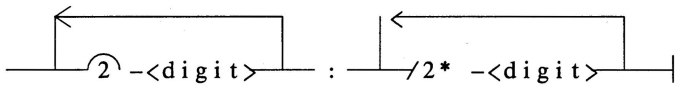
<starttime specification>



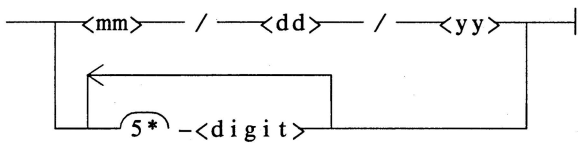
<starttime spec>



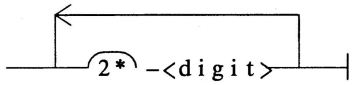
<time>



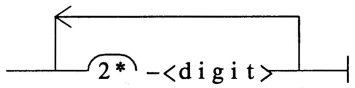
<date>



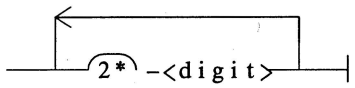
<mm>



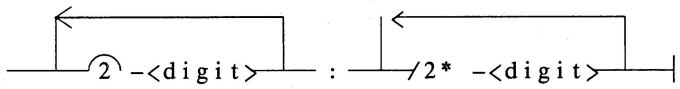
<dd>



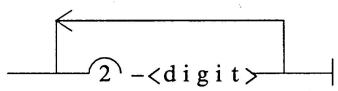
<yy>



<time interval>



<day interval>



Semantics

When a <starttime specification> is included in a <job attribute specification>, the job is handled in the normal fashion, except it is not selected to run until either the current time is greater than or equal to the specified start time or the job is FSed. (Refer to FS in the B 5000/B 6000/B 7000 Series Operator Display Terminal [ODT] Reference Manual.)

The WFL compiler determines the absolute time and date at which a job should begin execution from the <starttime spec>.

<time> is the time of day on a 24-hour clock. <time> and <time interval> are of the form HH:MM, where HH specifies the hour (or number of hours) and MM specifies the minute (or number of minutes). HH must be less than 24, and MM must be less than 60.

If a <time interval> is specified, that <time interval> is added to the current time.

If a <day interval> is specified, that number of days is added to the current date.

If <time> is used without a <date> or <day interval>, the current date is used.

If the JOBDESC file is removed, jobs with <starttime specification>s are also removed.

Examples:

The following job begins execution after 10:00 P.M. on March 20, 1981:

```
?BEGIN JOB EXAMPLE1;
   CLASS = 0;
   STARTTIME = 22:00 ON 03/20/81;
?END JOB.
```

The following job begins execution a minimum of one hour and 30 minutes after entering the system:

```
?BEGIN JOB EXAMPLE2;
   STARTTIME = +1:30;
?END JOB.
```

The following job is executed on host BLUE and begins execution after 11:00 A.M. according to the system clock on host BLUE:

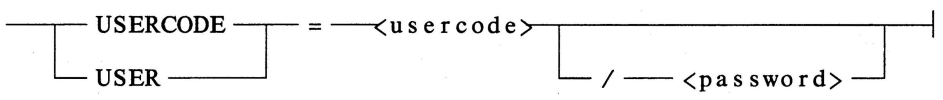
```
?AT BLUE BEGIN JOB;
   STARTTIME = 11:00;
?END JOB.
```

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES

Usercode Specification

Syntax

<usercode specification>



Semantics

The <usercode specification> is used to assign a usercode and password to a job. This usercode is retained across a Halt/Load. The <usercode> and <password> may only be <name constant>s.

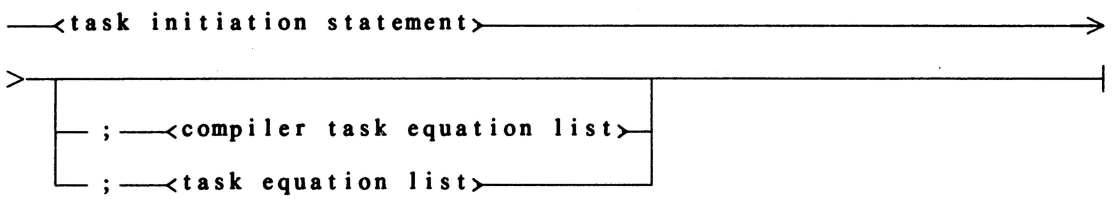
Examples

```
USER     = MYCODE

USER     = A/B

USERCODE = X/Y
```

TASK INITIATION

Syntax

<task initiation>



Semantics

Task initiation statements are used to start user programs and system functions in separate tasks. The task initiation statements are as follows:

1. COMPILE or BIND statement.
2. RUN statement.
3. COPY and ADD statements.
4. START statement.
5. LOG statement.
6. PB statement.
7. SCR statement.
8. PROCESS statement

A task is any process that has its own stack. The PROCESS statement initiates the task asynchronously; that is, the job stack executes concurrently with the task stack. All other task initiation statements run synchronously; that is, the execution of the job is suspended until the task is completed.

The START statement is not a task initiation statement but a job initiation statement. Therefore, the job runs asynchronously with the job that contained the associated START statement.

A task variable may be attached to the task in order to monitor and control the execution of the task. A task variable may be attached to only one task at a time. Therefore, the following statement is permissible:

```
RUN X[T];
COPY A TO B[T];
```

However, a run-time error results from the following statement:

```
PROCESS COMPILE X WITH ALGOL [T];
RUN Y[T];
```

A task equation allows specification of task attributes and file equation for any task at either compile- or run-time. The syntax for task equation is explained in the following subsection entitled Task Equation.

A file equation assigns a data file to serve one of three functions:

1. As an input file into which the program writes data.

2. As an output file into which the program writes data.

3. As both an input and output file.

Because many choices are presented in creating and using files, each file accessed by a program must be described to the system. The syntax for file equation is explained in a subsequent subsection entitled File Equation.

<file attribute assignment>s and <task attribute assignment>s are associated with the <task id> attached to a task. When the <task id> is attached to another task, the most recent assignments associated with the <task id> are recalled. The syntaxes for <task attribute assignment> and <file attribute assignment> are given under Task Attribute Assignment and File Attribute Assignment in this section.

Tasks may reference two types of data: (1) internal (entered as part of the job), and (2) external (entered separately from the job and read directly from the card reader). The syntax for data specification is explained in the subsection entitled Data Specification in this section.

Examples

```
    RUN X;
    RUN Y;
% PROGRAM X RUNS; WHEN IT IS FINISHED, PROGRAM Y RUNS.

    PROCESS X;
    RUN Y;
% PROGRAMS X AND Y RUN IN PARALLEL.
```

In the example below, both SYSTEM/CARDLINE runs create punched card output. The first time SYSTEM/CARDLINE is run, file LINE has an associated KIND of PUNCH. This label equation is also associated with the task variable T. The file titled INPUT1 is output to the card punch. When SYSTEM/CARDLINE is run the second time, the label equation associated with T is used; hence, file LINE still has a KIND of PUNCH, and the file titled INPUT2 is output to the card punch.

When the STATUS task attribute is set to NEVERUSED, no attribute assignments are associated with a task variable. In the example below, the statement

```
T(STATUS=NEVERUSED);
```

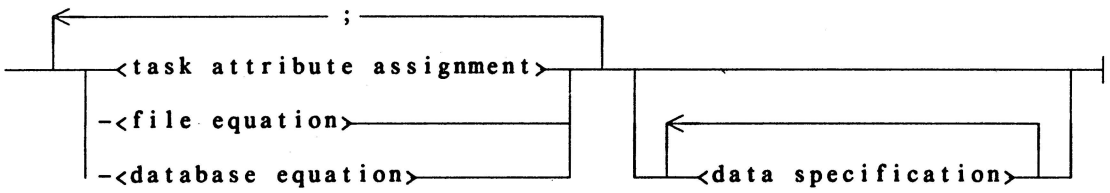
may be inserted between the two RUN statements. The second run of SYSTEM/CARDLINE then generates line printer output.

```
<i>BEGIN JOB TASK/TEST;
 TASK T;
<i>RUN SYSTEM/CARDLINE[T];
 FILE CARD (KIND=DISK,TITLE=INPUT1);
 FILE LINE (KIND=PUNCH);
 RUN SYSTEM/CARDLINE[T];
 FILE CARD (KIND=DISK,TITLE=INPUT2);
<i>END JOB
```

Task Equation

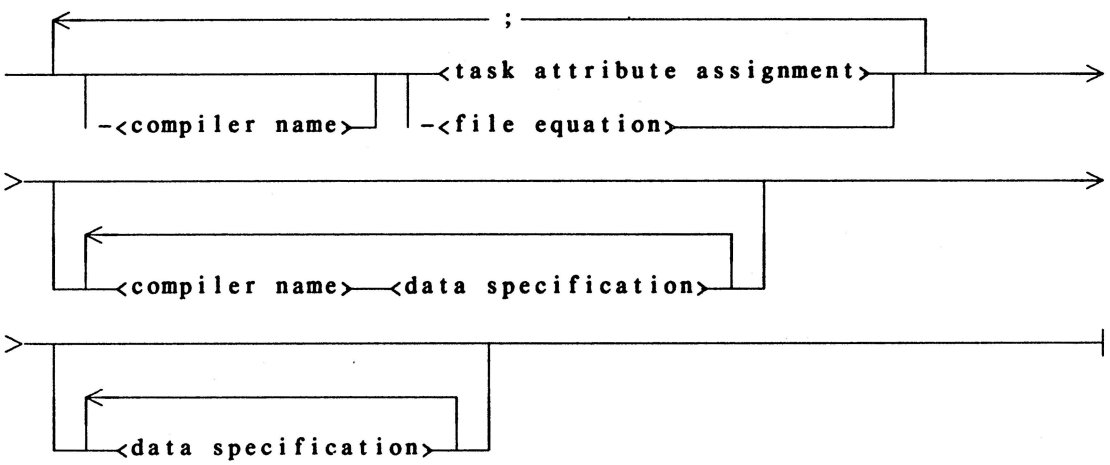
Syntax

<task equation list>



The COMPILE (or BIND) statement also has an optional <compiler task equation list>. The general form is as follows:

<compiler task equation list>



Semantics

Task initiation statements (excluding <compile or bind statement>) allow an optional <task equation list>.

Only the COMPILE and BIND statements may be followed by a <compiler task equation list> containing the optional <compiler name>. Although <task attribute assignment> and <file equation> specifications for the compiler and the source program may be intermixed, the <data specification>s for the compiler must precede all <data specification>s for the source program. All expressions in <task attribute assignment>s and <file attribute assignment>s (in the <file equation part>) for the source program are evaluated prior to compilation. The values obtained are stored in the code file as if they had been specified as constants.

Task attributes and file equation can be specified at two times:

1. When the code file is compiled.

2. When the task is initiated.

The data records for the compiler should follow the <compiler name> <data specification> statement. Data for execution should follow the <data specification>. If these sources conflict, the attributes specified at initiation time override.

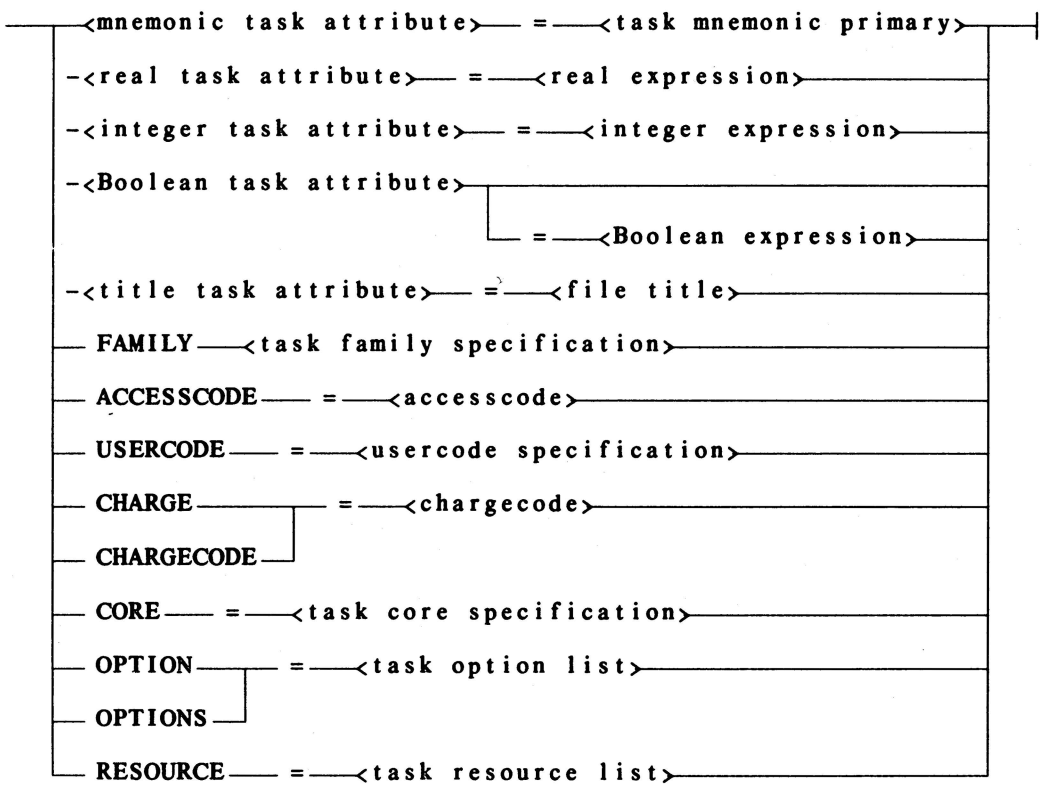
Example

```
COMPILE X ALGOL LIBRARY;
     ALGOL PRIORITY=50;    % SETS PRIORITY OF ALGOL COMPILATION
     PRIORITY=60;          % SETS PRIORITY IN CODE FILE X
RUN X;                     % RUNS AT PRIORITY 60
RUN X;
     PRIORITY=70;          % OVERRIDES COMPILED-IN PRIORITY
```

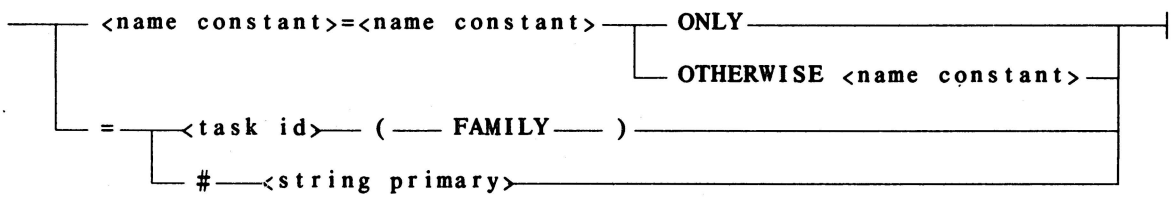
Task Attribute Assignment

Syntax

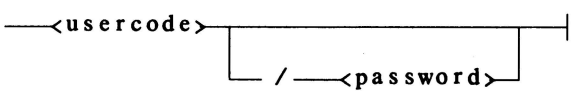
<task attribute assignment>



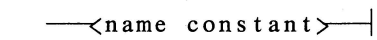
<task family specification>



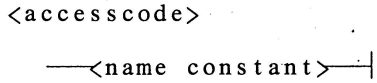
<usercode specification>



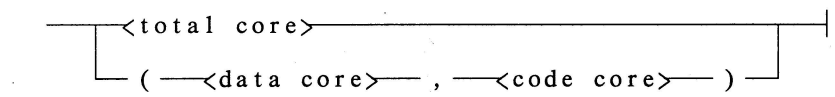
<chargecode>



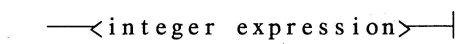
<accesscode>



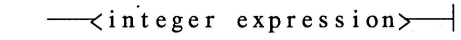
<task core specification>



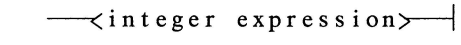
<total core>



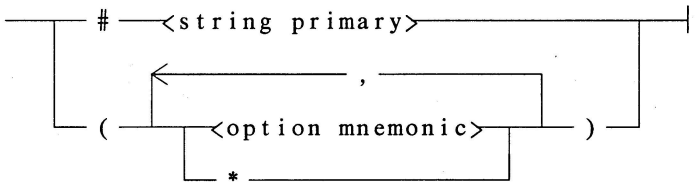
<data core>



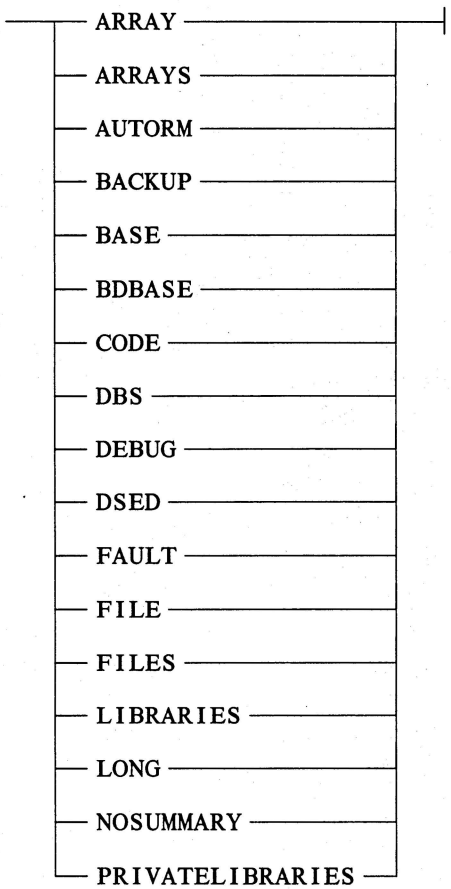
<code core>



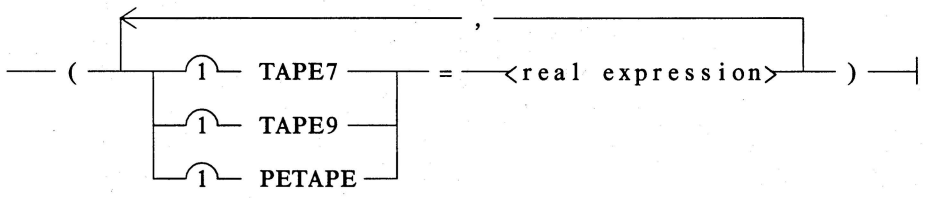
<task option list>



<option mnemonic>



<task resource list>



Semantics

<task attribute assignment> is used in <task declaration> statements, <task assignment statement>s, and <task equation list>s. Attribute values in <task declaration> statements must be constants or constant expressions.

Task attributes allow the job to monitor and control the execution of tasks. Misusing an attribute (for example, setting a read-only attribute) results in a run-time error. For details on the meanings and uses of task attributes, refer to Appendix A in this manual.

The <option mnemonic> * in the <task option list> specifies that any <option mnemonic>s provided with a later assignment are to be merged with the option mnemonics specified by this assignment. This mnemonic is particularly useful when assigned to a code file being compiled to LIBRARY.

The #<string primary> syntax may be used to specify the <task option list> in a <task attribute assignment> of the form OPTION=<task option list>. When used in this form, the value of <string primary> may not contain parentheses.

An accesscode may be applied to a job or task. Supplying an accesscode for a task does not change the accesscode of a job. If no accesscode is supplied for a task, the task inherits the accesscode of the job.

The <accesscode>/<password> is suppressed in the job summary printout. The SECURITY WFL statement accepts CONTROLLED <file name>.

A global task variable maintains its values throughout the scope of the job. Therefore, task T can only be attached to one task at any given time and may not be reused until that task has terminated.

Examples

```
COMPILE X FORTRAN LIBRARY;
OPTION=(*,FAULT);
     .
     .
     .
RUN X;                          % X RUNS WITH OPTION=(FAULT)
     .
     .
     .
RUN X;                          % X RUNS WITH OPTION = (FAULT,
OPTION = (ARRAYS,FILES);        %   ARRAYS,FILES)
```

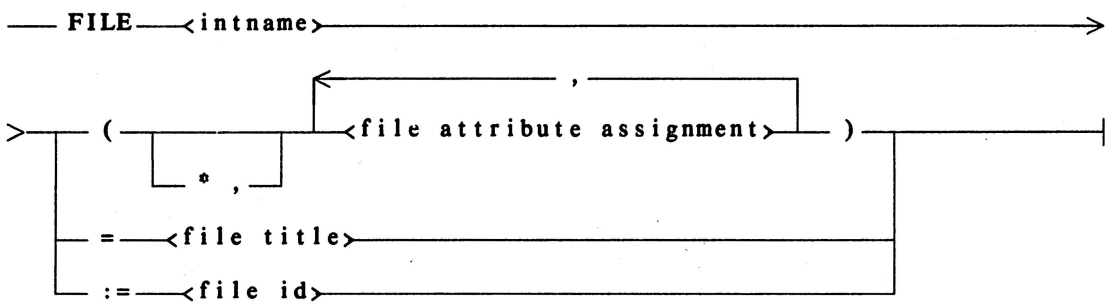
The following example illustrates use of the #<string primary> syntax to specify the <task option list>.

```
.
.
.
String S;
S:="LONG,FAULT";
RUN X;
OPTIONS=#S;            % X RUNS WITH THE OPTIONS LONG AND FAULT.
.
.
.
```

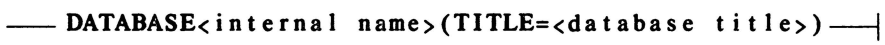
File and Database Equations

Syntax

<file equation>



<database equation>



Semantics

The <intname> is a declared <file id> in the object program to which the <file equation> is applied.

<file equation> specifies changes to the various file identifiers declared in the program that is being initiated.

If the object program to which a <file equation> is applied opens a file whose INTNAME is <intname>, the <file attribute assignment>s specified in the <file equation> are merged with the attributes specified in the program. If the same attribute is specified in both the <file equation> and the program, the <file attribute assignment> specified in the <file equation> takes precedence. If a file that is file equated is not opened by the program, the <file equation> has no effect.

<file equation> is used in <task declaration> statements, <task assignment statement>s, and <task equation list>s.

If more than one <file equation> is specified for the same internal file name, only the last <file equation> is used. If file attribute information from more than one <file equation> statement is to be merged, then the second and subsequent <file equation> statements must use an asterisk as the first attribute in the <file attribute assignment> list.

<database equation> statements may be included in WFL job decks. A <database title> in a host language <database declaration> may be overridden through WFL during either program compilation or execution.

Example

In this example, only the second <file equation> statement is used. The KIND and TITLE attributes are not used.

```
FILE X1(KIND=DISK,TITLE=TEST/X1);
FILE X1(MAXRECSIZE=20,UNITS=CHARACTERS);
```

The attributes from both statements are merged. The KIND and TITLE <file attribute>s are both used.

```
FILE X1(KIND=DISK);
FILE X1(*,TITLE=TEST/X1);
```

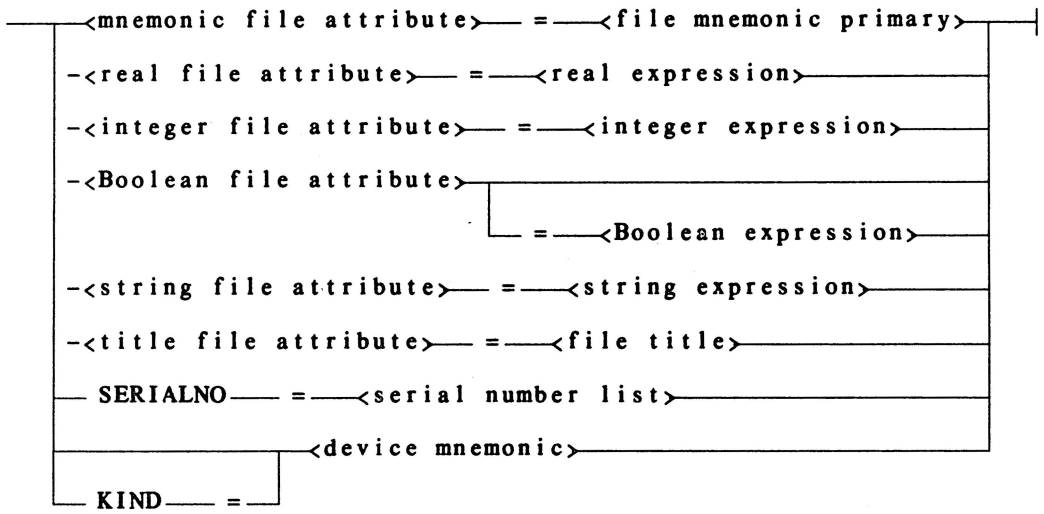
An example of a database equation statement follows:

```
DATABASE TESTDB(TITLE=MYDB)
```

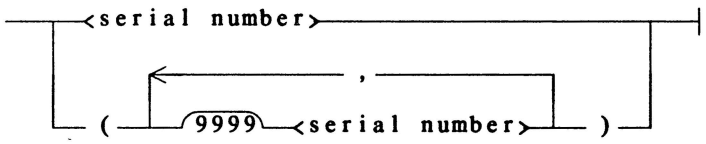
File Attribute Assignment

Syntax

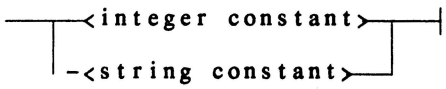
<file attribute assignment>



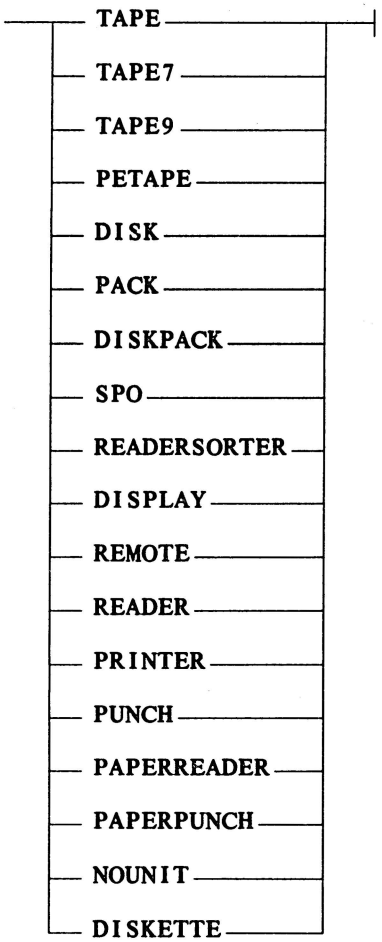
<serial number list>



<serial number>



<device mnemonic>



Semantics

<File attribute assignment>is used in <file declaration> statements, <file assignment statement>s, and <file equation> statements. Attribute values in <file declaration> statements must be constants or constant expressions.

Boolean, real, integer, string, mnemonic, and title <file attribute>s as well as <file mnemonic>s are described in the B 7000/B 6000 Series I/O Subsystem Reference Manual.

If a <serial number> is an <integer constant>, it must have a positive value less than or equal to 999999. The actual <serial number> is formed by the following <string expression>:

```
STRING(<integer constant>, 6)
```

If a <serial number> is a <string constant>, it must contain at least one and not more than six characters and must contain only alphanumeric characters. The actual <serial number> is formed by the following <string expression>:

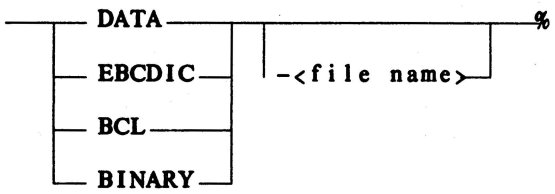
```
TAKE(<string constant> & "      ", 6)
```

If the restrictions on the form of the <serial number> are not satisfied, a run-time error is given.

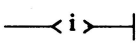
A new KIND, DISKETTE, was added to WFL to allow use of industry compatible minidisks (diskettes). The WFL compiler allows an explicit KIND=DISKETTE statement only if the volume name is also DISKETTE. (Named DISKETTES are not allowed.)

DATA Specification

Syntax



— followed by a deck of EBCDIC, BCL or Binary card images —%



Semantics

Tasks may reference two types of data decks: (1) internal (entered as as part of the job) and (2) external (entered separately from the job and read directly from the card reader). The first card of an external deck must contain a DATA specification with the name of the data deck. The deck is terminated by an END card.

System option 7 (CDONLY), which is controlled from the ODT, determines whether or not external decks may be referenced. If the option is RESET, then (1) any task that cannot find a card deck in its jobfile hangs on a NO FILE condition (requiring operator intervention) until the deck appears in a physical reader or the task is DSed, and (2) if a DATA specification is read which is external to any job, the card reader is labeled and made available to user programs.

If the option is set, then (1) any task that cannot find a card deck in its job file is DSed by the system, and (2) if a DATA specification is read which is external to any job, a syntax error occurs and the deck is flushed from the reader.

A task may read internal decks that are located between its task invocation statement and the next statement (this type is referred to as a local deck) or <global data deck>s located in the <job declaration statement list>.

A global deck must have a name, but local decks may be unnamed. When a task tries to open a card file, the first unread local deck with no name or the correct name is sought. If a local deck cannot be found, the first global deck with the correct name is sought. (Local decks are read only once per task stack execution, but global decks may be referenced many times. Each task stack opening a data deck starts reading at the beginning of the deck.)

WFL displays the following warning whenever a BCL deck is used:

BCL CARD DECKS ARE NOT PORTABLE TO EBCDIC MACHINES

The statement following a BCL or EBCDIC data deck must start with an invalid character in column 1. BINARY decks must be terminated by a binary END (BEND) card. The BEND card is only a data terminator; it does not act as an END card for the job.

A <compiler name> must precede DATA, EBCDIC, BCL, or BINARY on decks to be read by a compiler. Thus, a compiler is prevented from reading data decks intended for the "go part", and the go part is prevented from reading decks intended for a compiler. WFL requires that compiler data decks precede the go part data decks and that all attribute specifications precede any data decks. See COMPILE or BIND Statement in Section 6 of this manual for an explanation of the go part.

Examples

```
<i>COMPILE X ALGOL;
   COMPILER PRINTLIMIT=1000;
   PRINTLIMIT=2000;
<i>COMPILER DATA
      ....      % DATA FOR COMPILER
<i>DATA
      ....      % DATA FOR EXECUTION
<i>
   ....


<i> JOB X;

   BEGIN
   DATA GLOBAL/DECK1;
      .
      .
      .

<i> BINARY GLOBAL/DECK2;
      .
      .
      .
   (BEND CARD)

<i> RUN X;
   BCL
   .
   .
   .

<i> COMPILE A FORTRAN;
   FORTRAN DATA
   .
   .
   .

<i> DATA
   .
   .
   .

<i> END JOB
```

The following ALGOL program makes 20 BEND cards:

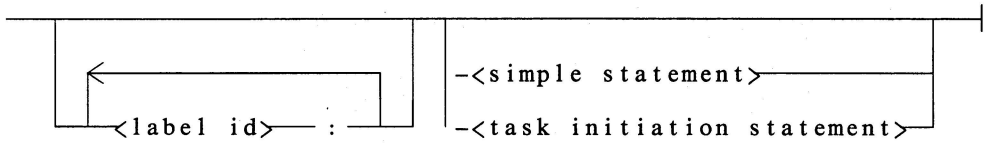
```
<i> BEGIN JOB;
    COMPILE MAKE/BEND/CARDS WITH ALGOL GO;
    COMPILER DATA CARD
    BEGIN
      FILE P(KIND = PUNCH, EXTMODE=BINARY);
      THRU 20 DO
      BEGIN
        P.OPEN := TRUE;
        CLOSE(P);
      END;
    END.
<i> END JOB.
```

6. STATEMENTS

INTRODUCTION

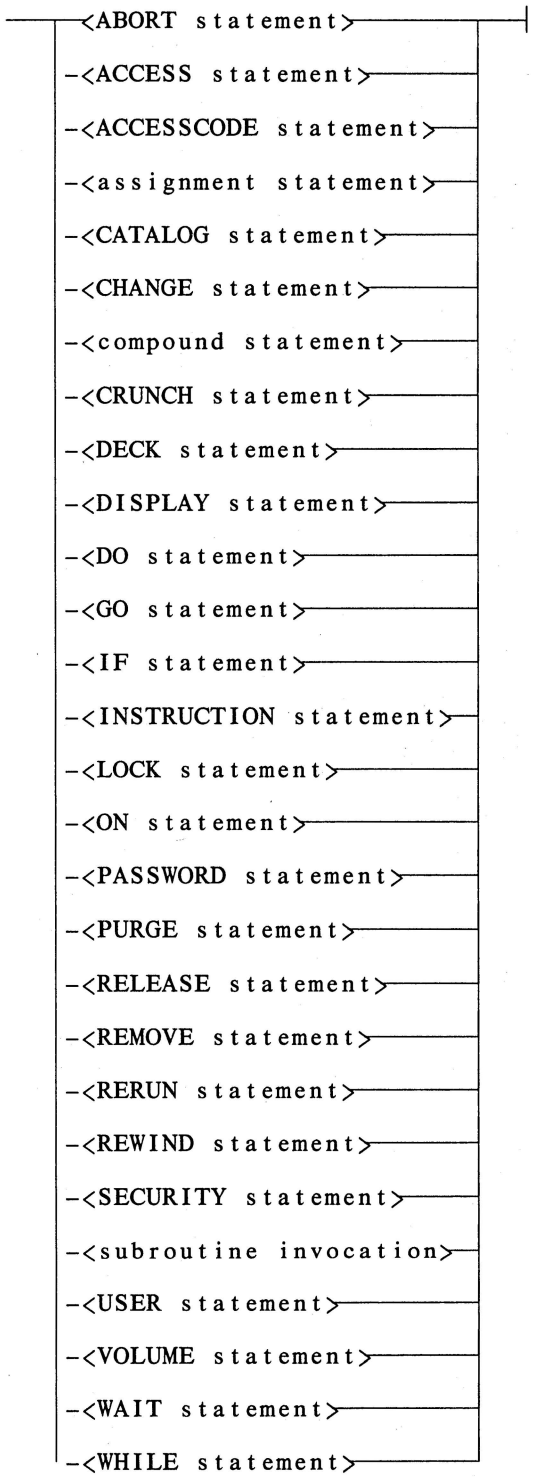
A statement defines the process flow through the use of control cards that manipulate the previously declared variables.

<statement>



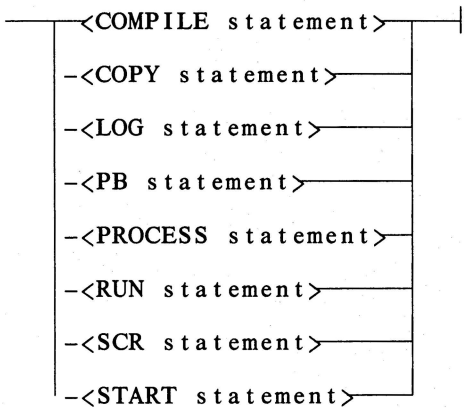
The path through the syntax that does not cross any metalinguistic token is a null statement. The null statement is used if no action is desired.

<simple statement>



THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

<task initiation statement>

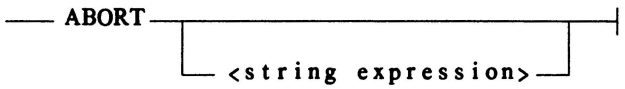


The statements described in this manual are divided into two groups: (1) all statements except library maintenance statements are presented in alphabetical order, followed by (2) the library maintenance statements, also presented in alphabetical order.

No <statement> may be nested to a depth greater than 19 levels. A <statement> specified in an ON statement, an IF statement within an IF statement, and <statement>s within a compound statement are all examples of nested statements.

ABORT Statement

Syntax



Semantics

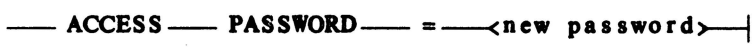
The ABORT statement causes the job task and any tasks initiated by the job task to be discontinued. The string specified in the statement is displayed prior to the abort.

Examples

```
ABORT

ABORT "THIS JOB HAS BEEN ABORTED"

IF T(VALUE)=5 THEN ABORT
```

ACCESS Statement

Syntax



Semantics

The ACCESS statement is used to change the ACCESSCODE password of the current job.

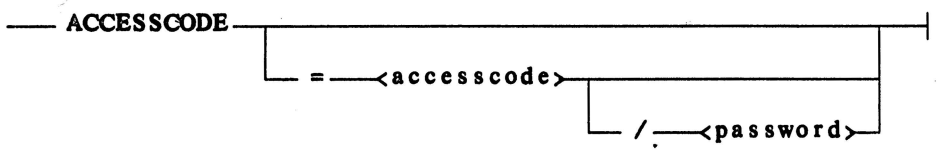
At execution, the job must have a usercode and a valid ACCESSCODE. The password for the current ACCESSCODE is changed to <new password>.

Example

```
ACCESS PASSWORD = ENTER
```

ACCESSCODE Statement

Syntax



Semantics

The ACCESSCODE form of this statement specifies that the job or task has no accesscode.

The ACCESSCODE=<accesscode> syntax specifies the accesscode for the job or task. The <accesscode>/<password> is validated in the USERDATAFILE.

An accesscode may be applied to a job or task. If no accesscode is supplied for a task, the task inherits the accesscode of the job. WFL accepts the ACCESSCODE task attribute whenever it is supplied with a task. Supplying an accesscode for a task does not change the accesscode of the job.

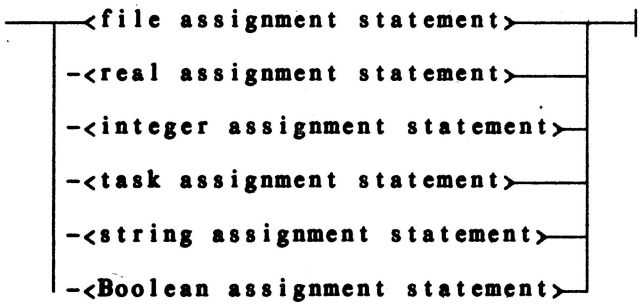
The <accesscode>/<password> is suppressed in the job summary printout.

Example

```
ACCESSCODE = OPEN/SESAME
```

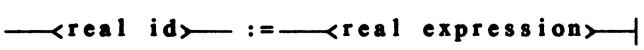
Assignment Statement

Syntax

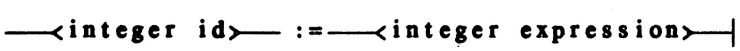


<file assignment statement>

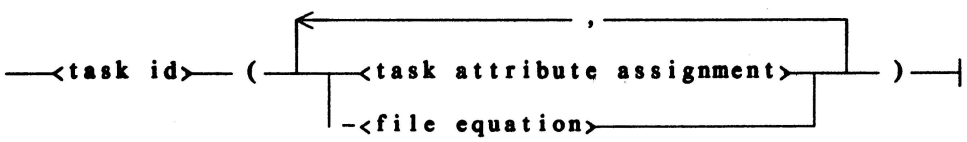
<real assignment statement>



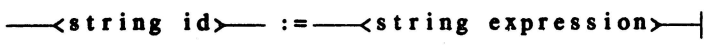
<integer assignment statement>



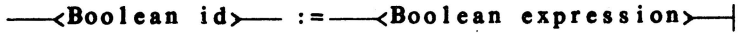
<task assignment statement>



<string assignment statement>



<Boolean assignment statement>



Semantics

The assignment statement assigns values to declared variables.

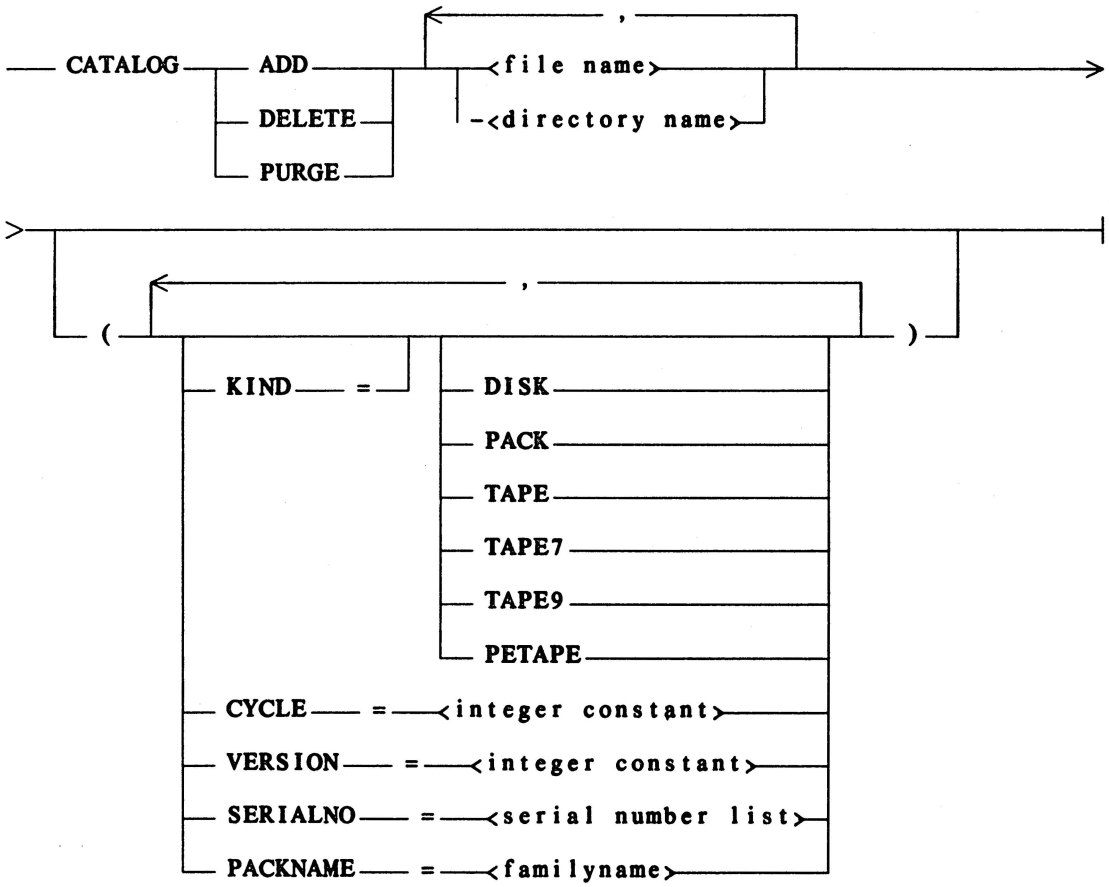
Example

In the following example, a task, file, real, integer, string, and Boolean are declared in the first group of variables. The statements in the second group assign values to the declared variables.

```
TASK T;
FILE F;
REAL R;
INTEGER I;
STRING S;
BOOLEAN B;
.
.
.
T(PRIORITY=70,MAXCARDS=200);        % Task Attribute Assignment
F(AREASIZE=1008,FLEXIBLE);          % File Attribute Assignment
R:=17.5;
I:=INTEGER(R);
S:="ABC"&STRING(I,*);
B:=FILE A/B ON PACK IS RESIDENT;
.
.
.
```

CATALOG Statement

Syntax



Semantics

The CATALOG statement applies only to a cataloging system. All available versions of a file (particularly the most current version) are acknowledged. Also, information regarding backup copies of available versions of a file is maintained.

CATALOG ADD is used to mark file entries as CATALOGED. CATALOG DELETE is used to delete specific entries from the catalog block that passes the appropriate file attributes used to select the file entry to be deleted. A CATALOG PURGE is used to remove all backup information from the catalog block for a particular file; the file entry is marked NOT CATALOGED.

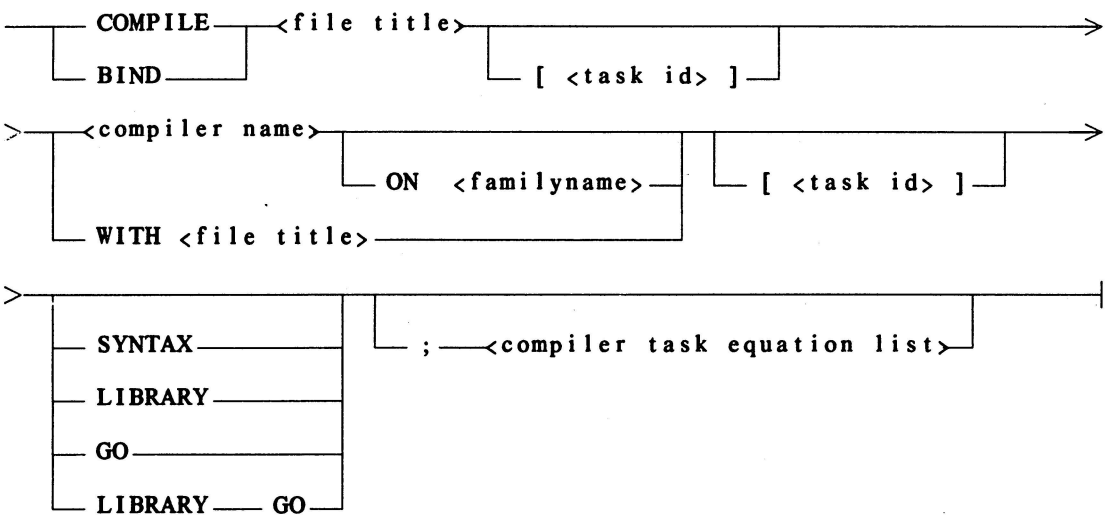
The CATALOG statement does not recognize the special volume name DISKETTE; if DISKETTE is specified, the statement is flagged with the message: NOT IMPLEMENTED YET.

Examples

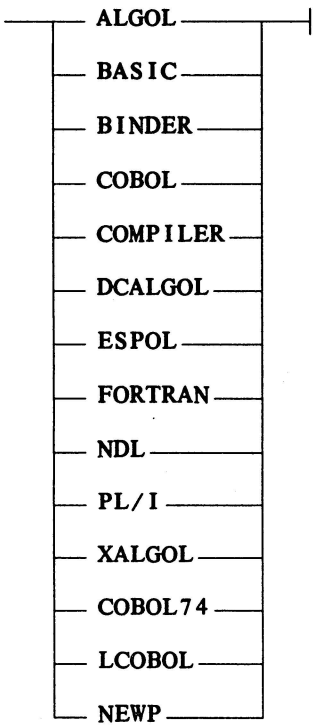
```
CATALOG ADD FILEA (KIND=PACK, SERIALNO=123456)

CATALOG DELETE=(KIND=TAPE9,VERSION=12,CYCLE=5)

CATALOG PURGE FILEA,FILEB(KIND=PACK)
```

COMPILE or BIND Statement

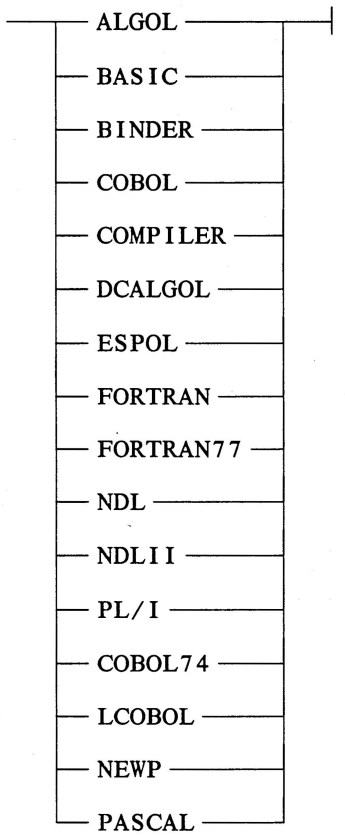
Syntax



<compiler name>



<compiler name>



THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES

Semantics

The COMPILE statement is used to initiate compilation. The first <file title> is the name of the resulting code file. The compiler is specified either by using a recognized system compiler or by preceding the file name with the word WITH. If the resulting file is not a compiler code file, the job is discontinued. The disposition of the code file is given following the compiler name, as follows:

GO	The code file is executed if no syntax errors are present. This disposition is the default condition.
LIBRARY	The code file is entered in the directory if no syntax errors are present.
SYNTAX	The code file is compiled for syntax only.
LIBRARY GO	If no syntax errors are present, the code file is entered in the directory and executed.

If a <task id> is specified following the code file name and the GO disposition is specified, the task variable is attached to the execution of the code file. This execution of the code file is termed the "go part" of the compilation.

The task variable specified following the compiler name is attached to the compilation. Task attributes and data decks may be assigned to the compiler by prefacing them by the word COMPILER or the name of any compiler. (Refer to Task Equation in Section 5 for the syntactic definition of <compiler task equation list>.)

Attributes (but not data decks) which are not preceded by COMPILER or a <compiler name> are permanently attached to the code file. These values are assigned to the task whenever the compiled program is executed unless the attribute values are overridden by run-time task attribute values (refer to the RUN statement in this section).

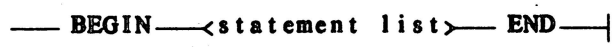
Local data decks (those that follow a COMPILE or BIND statement) are reread if the initiating statement is executed more than one time (if, for example, the data deck is in a subroutine or a loop).

Examples

```
COMPILE X/Y COBOL LIBRARY GO;
     COBOL FILE CARD(TITLE = C/D, KIND = DISK);
     COBOL PRIORITY = 55;
     FILE F(TITLE=Y/Z)

COMPILE A WITH PL/I SYNTAX

COMPILE X ALGOL

COMPILE B FORTRAN ON PACK

BIND X/Y BINDER LIBRARY AND GO

COMPILE A/B WITH ALGOL;
     COBOL PUNCHLIMIT = 100;
     COMPILER PRINTLIMIT = 130;
     ALGOL DATA
        ....
     <i> ....
```

Compound Statement

Syntax



Semantics

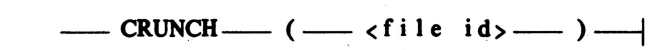
A compound statement allows one or more statements to be grouped as a logical entity. (Refer to JOB SYNTAX AND STRUCTURE in Section 5 for the syntactic definition of <statement list>.)

Examples

```
BEGIN RUN X; END

BEGIN RUN Y; FILE F1 := GLOBAL;DISPLAY"#|?$"; END
```

CRUNCH Statement

Syntax



Semantics

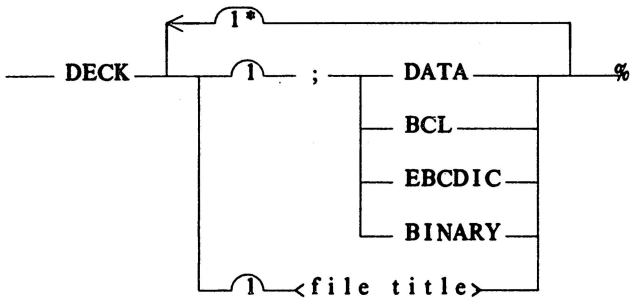
The CRUNCH statement closes and CRUNCHes a file. CRUNCHing causes the unused portion of the last row (beyond the end-of-file indicator) of disk space to be returned to the system. In order to be CRUNCHed, the file must be a disk file and, as a result of being CRUNCHed, can no longer be expanded. However, the file can be written to a location inside the end-of-file limit (last block boundary).

Examples

```
CRUNCH (LONGFILE)

CRUNCH (BITSFILE)
```

DECK Statement

Syntax



—— followed by a deck of card images ——%

Semantics

The DECK statement copies card images from the WFL job deck to disk (or disk pack). The <file title> given in the statement specifies the title of the resulting disk file. The file title may be placed after the word DECK or the word DATA. Only one DECK statement is allowed in a job.

A standard binary END (BEND) card is required to terminate BINARY data decks. This BEND card is a data terminator only; the job must still be terminated by an <i>END JOB.

The disk file is created by the WFL compiler when the job deck is being read in the system. Once WFL has locked the file onto the system, this action cannot be reversed if syntax errors are discovered later in the job. Thus, if a DECK statement appears, it must be the only statement in a job.

WFL allows only the attributes USERCODE (or USER), CHARGE (or CHARGECODE), FAMILY, DESTNAME, NAME, and CLASS (or QUEUE) to appear in the <job attribute specification> list of a job containing a DECK statement. (The CLASS attribute is ignored). Any other attribute appearing in the <job attribute specification> list results in a syntax error and prevents creation of the disk file.

The file attributes of the resulting disk file are as follows:

KIND	= DISK or PACK
FILEKIND	= DATA
SAVEFACTOR	= 999
MAXRECSIZE	= 10 for BCL
	= 15 for DATA
	= 15 for EBCDIC
	= 20 for BINARY
BLOCKSIZE	= 420 for BCL
	= 420 for EBCDIC
	= 480 for BINARY
AREASIZE	= 1512 for BCL
	= 1008 for EBCDIC
	= 756 for BINARY
EXTMODE	= BCL for BCL
	= EBCDIC for DATA
	= EBCDIC for EBCDIC
	= SINGLE for BINARY

If the SYNTAX or NEWSOURCE <disposition> is specified in the BEGIN JOB statement, the DECK statement executes properly. SYNTAX causes the job to be checked, and no new disk file is created. NEWSOURCE causes the entire job deck to be saved as a disk file; in this case, the data must not be BINARY.

Example

```
BEGIN JOB DECK/STATEMENT;
DECK SAMPLE1;DATA
  DATA IN THIS DECK
       .
       .
       .
?END JOB

BEGIN JOB DECK/STATEMENT;
DECK;DATA SAMPLE2
  DATA IN THIS DECK
       .
       .
       .
?END JOB
```

DISPLAY Statement

Syntax

Semantics

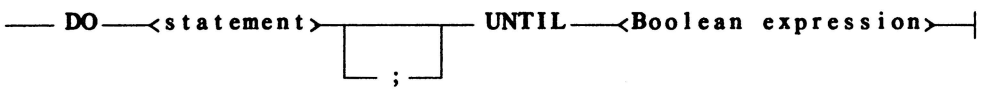
The DISPLAY statement causes a message to be displayed on the ODT and placed in the job log.

Example

```
DISPLAY "HI THERE"

DISPLAY F1 & "DID NOT COMPILE"
```

DO Statement

Syntax



Semantics

The DO statement allows a job to execute a statement until a condition is TRUE.

The statement following the DO is executed, and then the <Boolean expression> is evaluated. If the expression is TRUE, control passes to the next executable statement; otherwise, the statement after the DO is re-executed.

Example

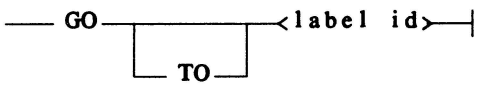
```
DO BEGIN A:=A+1; RUN X[T]; VALUE=A;
   END UNTIL T(VALUE)=4;
```

CAUTION

Care should be exercised to ensure that an unlimited number of RUNs are not initiated and that a <task id> is not reused before a previous occurrence terminates.

GO Statement

Syntax



Semantics

The GO statement unconditionally transfers control to a label.

Example

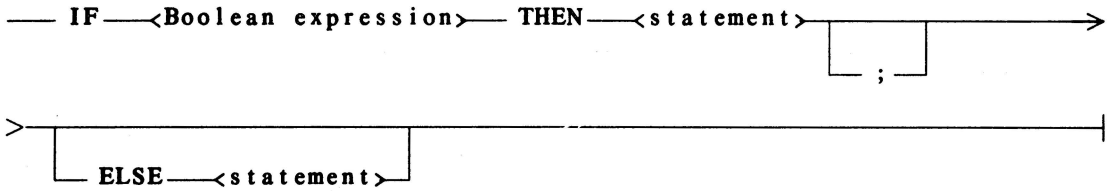
```
GO TO L;
    .
    .
    .
L: RUN X;
L1 : L2 : RUN Y;
GO L2;
```

CAUTION

When using the GO statement to create loops, care should be exercised to ensure that an unlimited number of RUNs are not initiated and that the same task identifier is not reused before a previous occurrence terminates.

IF Statement

Syntax



Semantics

The IF statement allows a conditional decision to be made based on the evaluation of a <Boolean expression>. The statement following THEN is executed if the condition is TRUE. If the condition is FALSE, control passes to the next executable statement. When the ELSE option is specified and the condition is FALSE, the statement following ELSE is executed.

Examples

```
IF T IS ACTIVE THEN GO TO L ELSE I:=1

IF T(VALUE)=5 THEN RUN X

IF FILE X/Y ISNT RESIDENT THEN
   DISPLAY "NO FILE X/Y"; ELSE RUN X/Y

IF B THEN GO XIT

IF I=J THEN BEGIN RUN X; RUN Y; END ELSE GO Z
```

CAUTION

When using the IF statement to create loops, care should be exercised to ensure that an unlimited number of RUNs are not initiated and that the same task identifier is not reused before a previous occurrence terminates.

INSTRUCTION Statement

Syntax

Semantics

The INSTRUCTION statement is used to supply job instructions to operators. <integer constant> may have the value 1 thru 63. If no value is provided for <integer constant>, a syntax error results.

As a WFL job executes, any INSTRUCTION statements encountered are stored in a table. As each INSTRUCTION statement is found, it is marked as the most current instruction until another one is found. At any point during execution of the job, any individual instruction may be displayed by number through the Instruction Block (IB) ODT message. If an instruction number is not specified, the system displays the most current instruction.

Examples

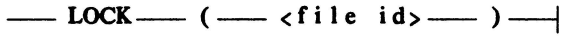
```
INSTRUCTION 5 DS MY JOB IF NO FILE A

INSTRUCTION 2 MOUNT TAPE TEST3
```

In the following example, the system needs tape TESTTAPE during execution of the COPY statement. If the operator asks for the most recent instruction, instruction 1 is displayed indicating where TESTTAPE can be found. Subsequently, the job needs files T17 and T17A; an instruction request at that point causes instruction 2 to be displayed with instructions regarding the action to be taken if T17 and T17A are not present.

```
?BEGIN JOB COMPILE/TESTS;
 FAMILY DISK = USERS OTHERWISE DISK;
    INSTRUCTION 1 TESTTAPE IS IN TAPE RACK 3.;
     COPY&COMPARE = FROM TESTTAPE TO USERS(PACK);
    INSTRUCTION 2 IF T17 OR T17A WERE NOT COPIED FROM TESTTAPE TO
       USERS, PLEASE DS THIS JOB AND LEAVE JK A NOTE.;
    COMPILE TEST/17 ALGOL;
       ALGOL FILE CARD(TITLE=T17, KIND=DISK);
       FILE F(TITLE=T17A);
    IF FILE TEST/17 ISNT RESIDENT THEN ABORT "**BAD COMPILE**";
?END JOB.
```

LOCK Statement

Syntax



Semantics

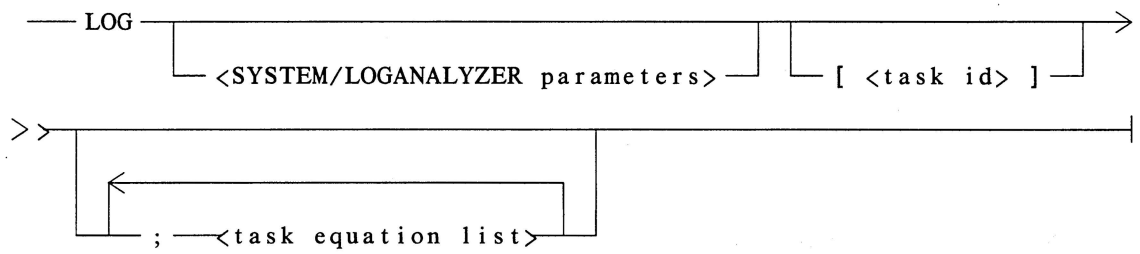
The LOCK statement closes the referenced file. If the file is on tape, the tape is rewound, and a system message is displayed which indicates that the reel must be removed and saved. If the file is not a disk or pack file, the unit is made inaccessible to the system and must be readied again manually. If the file is a disk file, it is retained as a permanent file on disk, and the file buffer areas are returned to the system.

Examples

```
LOCK (BIGFILE)

LOCK (TIOGLOBAL)
```

LOG Statement

Syntax



Semantics

The LOG statement initiates SYSTEM/LOGANALYZER and passes the <SYSTEM/LOGANALYZER parameters> to the SYSTEM/LOGANALYZER utility program. (Refer to Chapter 6 of the B 5000/B 6000/B 7000 Series System Software Operational Guide, Volume 2.)

NOTE

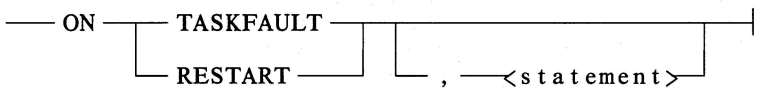
The WFL compiler does not analyze the parameters; this analysis is done by the SYSTEM/LOGANALYZER utility.

Examples

```
LOG

LOG "SYSTEM/SUMLOG" 2359 07/26/75 ALL SAVE [I]

LOG /123 2200 07/27/75 TO 0800 07/28/75 HL SAVE[T];PRINTLIMIT=50
```

ON Statement

Syntax



Semantics

The ON statement allows the job task to take user-specified action when (1) a task is abnormally terminated (ON TASKFAULT), or (2) the system is Halt/Loaded and the job must be restarted (ON RESTART).

Only one statement for each condition can be enabled at one time. Thus, any ON statement disables any previous ON statement for the same condition. If a subroutine executes an ON statement while running in the job, the ON statement disables any previous ON statement until the subroutine is finished or the subroutine ON condition is disabled.

The statement "ON TASKFAULT, <statement>;" enables the condition TASKFAULT. If any task is subsequently terminated abnormally or if a compilation is terminated for syntax errors, the <statement> is executed. These termination conditions include operator or programmatic discontinuations as well as program faults.

If a GO TO out of the ON TASKFAULT <statement> is not performed, the following actions occur:

1. Any taskfault interrupt that occurs during the execution of the ON TASKFAULT statement is queued.

2. When execution of the ON TASKFAULT statement is completed, action is taken on any taskfaults that were queued. A taskfault is selected, and a new invocation of the ON TASKFAULT statement is initiated.

3. The ON TASKFAULT statement is invoked for each taskfault that becomes queued. Therefore, the ON TASKFAULT statement continues execution until no taskfaults remain in the queue or until a GO TO out of the ON TASKFAULT statement is executed.

The statement "ON TASKFAULT;" disables the condition TASKFAULT. While this statement is in effect, an abnormal task termination has no effect on the job.

Similarly, the RESTART condition can be enabled and disabled by using the statements:

```
ON RESTART, <statement>;

ON RESTART;
```

Subroutine exit or entry of an ON RESTART <statement> returns the restart condition to the condition before the subroutine was called.

If the enabled statement does not execute a GO TO statement, the job resumes execution at the point it would have been if the statement had been disabled.

If the system is Halt/Loaded during the execution of a job, the job restarts at the most recent point at which no tasks were running. Typically, this point is just prior to the last task initiation. The values of all integer, real, and Boolean variables are reset to their values at the point of the restart. The attributes of task and file variables and the contents of string variables are not reinitialized. If the job uses this information, an ON RESTART statement may have to be included to reinitialize the task file and string variables. If the statement enabled by an ON statement is a compound statement, the information located in that compound statement is local and therefore unknown until the ON statement is invoked.

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES

Examples

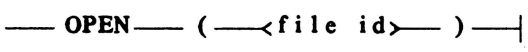
```
<i> BEGIN JOB X;
      SUBROUTINE SUB;
      BEGIN
        ON TASKFAULT,
          BEGIN
            DISPLAY "SUB TASKFAULT TAKEN";
            GO ERR;
          END; % END OF ON STATEMENT
        RUN X;% IF X ABORTS, "SUB TASKFAULT TAKEN" IS
              % DISPLAYED BY THE ON TASKFAULT;
        ON TASKFAULT;
        RUN Y;% IF Y ABORTS, "JOB TASKFAULT" IS DISPLAYED
      END; % END OF SUB
      RUN A; % IF A ABORTS, NO TASKFAULT IS EXECUTED
      ON TASKFAULT, DISPLAY "JOB TASKFAULT";
      RUN B; % IF B ABORTS, "JOB TASKFAULT" IS DISPLAYED
      SUB;
    ERR:
<i> END JOB
```

In the following example, task P2 is assumed to update a global file (G) created by P1. If a Halt/Load occurs while P2 is running, the job does not normally rerun P1 but reruns P2. In that case, P2 "double updates" any records that P2 had updated prior to the Halt/Load. Therefore, an ON RESTART statement is executed to assure that P1 is rerun and that the file is recreated.

```
<i> BEGIN JOB X;
      FILE G;
      ON RESTART, GO TO L;
L:    RUN P1; FILE F1:=G;
      RUN P2; FILE F2:=G;
<i> END JOB
```

OPEN Statement

Syntax



Semantics

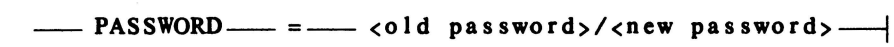
The OPEN statement is used to explicitly OPEN files.

Example

```
OPEN (FILEOUT)
```

PASSWORD Statement

Syntax



<old password>

<new password>

Semantics

The PASSWORD statement is used to change (in the SYSTEM/USERDATAFILE file) the password associated with the current usercode of the job.

NOTE

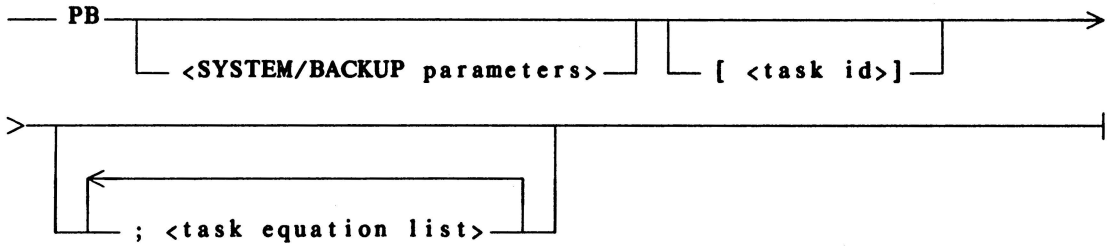
On the 3.2 release, pre-3.2 PASSWORD syntax is allowed, but a warning message is issued. On the 3.3 release, the old syntax will not be allowed.

Examples

```
PASSWORD = XYZ/ABC
```

PB Statement

Syntax



Semantics

The PB (Printer/Punch Backup) statement initiates SYSTEM/BACKUP, which is used to print or punch backup files. The PB statement passes the <SYSTEM/BACKUP parameters> to the SYSTEM/BACKUP utility program. (Refer to Chapter 1 of the B 7000/B 6000 Series System Software Operational Guide, Volume 1.)

NOTES

The WFL compiler does not analyze <SYSTEM/BACKUP parameters>; this analysis is done by the SYSTEM/BACKUP utility.

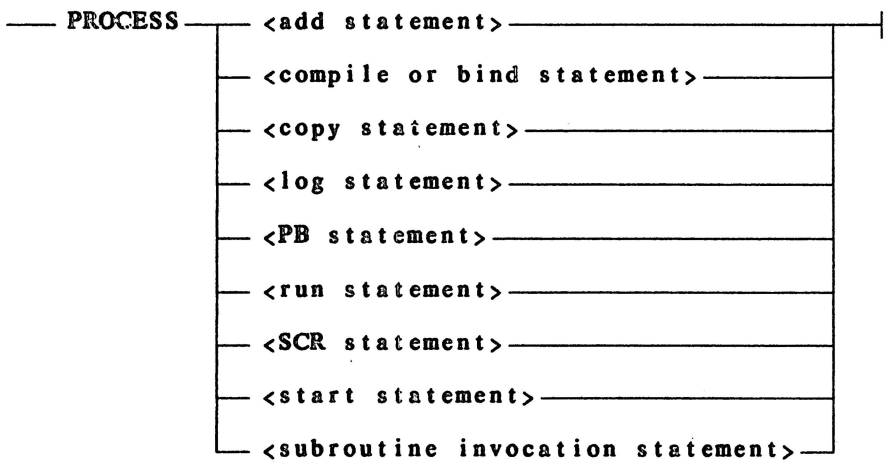
When this statement is entered from the ODT, a ?PB must be used to differentiate it from the PB ODT message.

Examples

```
PB

PB MT113 SAVE[I]

PB D123/FILEA LP11 COPIES=3 SAVE[T];PRINTLIMIT=500
```

PROCESS Statement

Syntax



Semantics

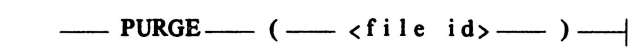
The PROCESS statement initiates tasks asynchronously. If a task variable is attached, the job stack can subsequently wait for the process to terminate (refer to WAIT Statement in this section), monitor its execution (refer to IF Statement in this section), or alter its execution (refer to Assignment Statement in this section). The job stack does not terminate until all asynchronous tasks have terminated.

Examples

```
PROCESS RUN X/Y (3,I) [T]

PROCESS SUB; CORE =100

PROCESS COPY A TO B [T]
```

PURGE Statement

Syntax



Semantics

The PURGE statement causes the file to be closed, purged, and released to the system. If the file is a permanent disk file, it is removed from the disk directory, and the disk space is returned to the system.

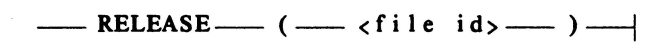
The file to be purged must first be explicitly opened by the OPEN (<file id>) statement.

Examples

```
PURGE (ALLFILES)

PURGE (ONEFILE)
```

RELEASE Statement

Syntax



Semantics

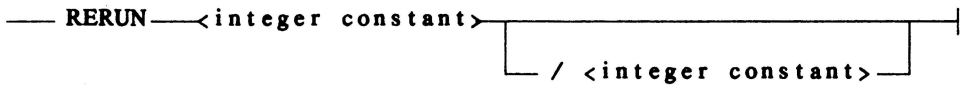
The RELEASE statement closes and releases a global file. The file buffer areas are returned to the system. If the file is a temporary file, the disk space is deallocated.

Examples

```
RELEASE (GLOBAL)

RELEASE (FILEA)
```

RERUN Statement

Syntax



Semantics

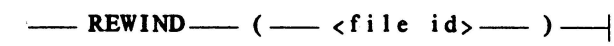
The RERUN statement allows a job to be programatically restarted at the last checkpoint. The first <integer constant> is the job number. The second <integer constant> indicates which checkpoint files are used for the restart. If the second number is omitted, the last checkpoint file for the indicated job number is restarted.

Examples

```
RERUN 3261

RERUN 3555/3
```

REWIND Statement

Syntax



Semantics

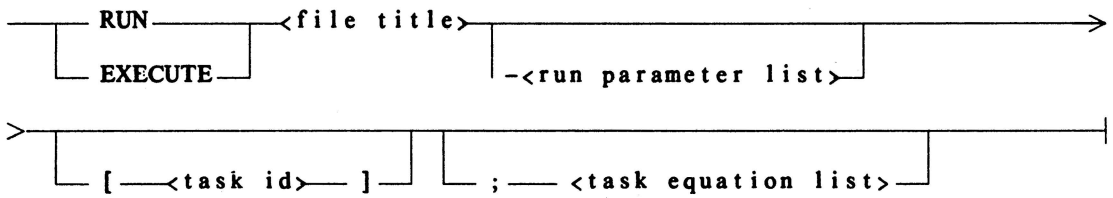
The REWIND statement causes the file to be closed. If the file is a paper-tape or magnetic-tape file, it is rewound. For disk files, the record pointer is reset to the first record of the file. The file buffer areas are returned to the system. The I/O unit remains under program control.

Examples

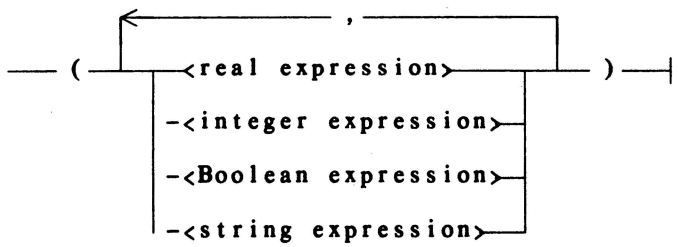
```
REWIND (GLOBAL)

REWIND (FILEA)
```

RUN Statement

Syntax



<run parameter list>



Semantics

The RUN statement initiates a previously created code file. If the <file title> is not a code file, the job is discontinued.

Parameter checking is done against the code file, and any mismatches cause the job to be discontinued. All parameters are passed by value (the value of a variable in the job stack cannot be changed by passing it to a task).

Strings are EBCDIC and are passed as arrays. For <string expressions>s, an extra NUL (48"00") character is added to the end of the string to allow the object program to determine the length of the string (the length is not explicitly passed).

A list of <task attribute assignment>s can be given which sets attributes in the task. These values override any attributes set when the code file was created. (Refer to TASK INITIATION in Section 5 and to COMPILE or BIND Statement in this section.)

The specifications for the program parameters depend on the language used for the program.

Examples

```
RUN X

RUN A/B [T]; MAXCARDS=500

RUN A/B (1,I,FALSE,3.14,OCTAL ("123"), "HI THERE") [T1];
      FILE C:=G;
      FILE F(BLOCKSIZE=10, KIND=PETAPE);
      FILE G(KIND=DISK);

RUN A/B; VALUE=I
```

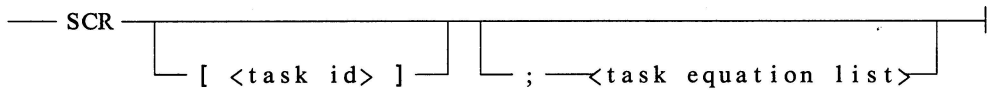
The examples below show the use of WFL-provided parameters for programs written in ALGOL, PL/I, and COBOL. (COBOL and PL/I cannot specify Boolean parameters.)

```
********************************   ALGOL   ********************************

<i>COMPILE ALG ALGOL LIBRARY
<i>ALGOL DATA
  $ SET LEVEL 2
  PROCEDURE D(WFLREAL,WFLSTRING,WFLBOOLEAN)
              VALUE WFLREAL,WFLBOOLEAN;
              REAL WFLREAL;
              ARRAY WFLSTRING[*];
              BOOLEAN WFLBOOLEAN;
  BEGIN
  EBCDIC ARRAY A[0:71];
     REPLACE A[0] BY "WFLREAL     = ",
                     WFLREAL FOR 10 DIGITS,
                     48"00";
     DISPLAY(A[0]);
     REPLACE A[0] BY "WFLSTRING   = ",
                     POINTER(WFLSTRING) FOR 30 UNTIL=48"00",
                     48"00";
     DISPLAY(A[0]);
     REPLACE A[0] BY "WFLBOOLEAN = ",
                     (IF WFLBOOLEAN THEN "TRUE  "
                                    ELSE "FALSE ") FOR 6,
                     48"00";
     DISPLAY(A[0]);
  END.

<i>RUN ALG (1234, "ABCD", FILE COB IS RESIDENT)
```

```
**********************************    PL/I     **********************************

<i>COMPILE PLI PL/I LIBRARY
<i>PL/I DATA
 D:PROCEDURE(WFLREAL<WFLSTRING);
 DECLARE WFLREAL DECIMAL FIXED(10,0),
         WFLSTRING CHAR(30) VARYING;
   DISPLAY ('WFLREAL    = ' || WFLREAL);
   DISPLAY ('WLFSTRING = ' || WFLSTRING);
 END D;

<i>RUN PLI (12*100+34,"ABCD")

**********************************    COBOL    **********************************

<i>COMPILE COB COBOL LIBRARY
<i>COBOL DATA
  100000$ SET ANSI74
  100100 ID D.
  100200 ENVIRONMENT DIVISION.
  100300 DATA DIVISION.
  100400 WORKING-STORAGE SECTION.
  100500 77  WFLREAL COMP PIC 9(11) RECEIVED BY REFERENCE.
  100600 01  WFLSTRING COMP RECEIVED BY REFERENCE WITH LOWER-BOUNDS.
  100700     03  FIVEWORDS PIC 9(11) OCCURS 5.
  100800 01  CHARMESSAGEPARAM REDEFINES WFLSTRING.
  100900     03  FILLER PIC X(30).
  101000 77  MESSAGESIZE COMP PIC 99 VALUE 30.
  101100 01  CHARMESSAGE
  101200         SIZE IS 1 TO 30 CHARACTERS DEPENDING ON MESSAGESIZE.
  101300     03  CHAR PIC X OCCURS 30.
  101400 PROCEDURE DIVISION USING WFLREAL, WFLSTRING.
  101500 P1.
  101600     EXAMINE CHARMESSAGEPARAM TALLYING UNTIL FIRST LOW-VALUE.
  101700     MOVE TALLY TO MESSAGESIZE.
  101800     MOVE CHARMESSAGEPARAM TO CHARMESSAGE.
  101900     DISPLAY "WFLREAL    = ", WFLREAL.
  102000     DISPLAY "WFLSTRING = ", CHARMESSAGE.
  102100     STOP RUN.


<i>RUN COB (1234, "AB" & "CD")
```

SCR Statement

Syntax



Semantics

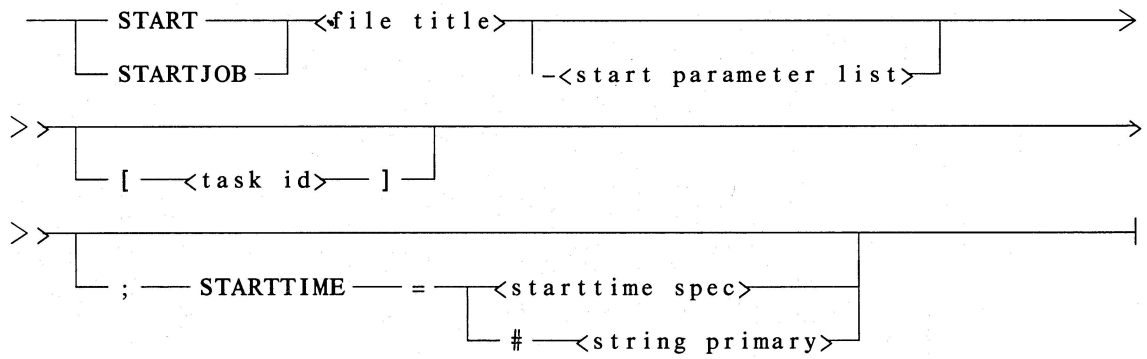
The SCR statement initiates the MCP on-line maintenance routine. The routine requires a data deck when initiated from the card reader. Refer to the B 6700/B 7700 On-Line Maintenance and Test (MAT) Language Information Manual.

Examples

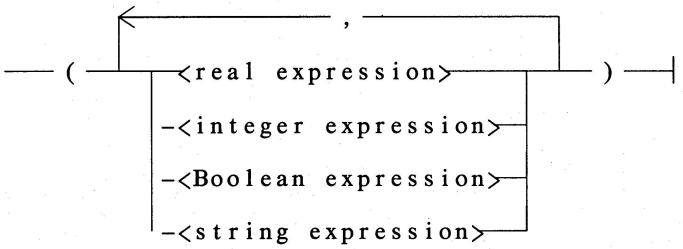
```
SCR

SCR [T]
```

START Statement

Syntax



<start parameter list>



Semantics

The START statement initiates a run of the WFL compiler as a coroutine named CONTROLCARD in the job in which the START statement appears. CONTROLCARD compiles the job to be started and causes subsequent insertion of the job in a job queue.

START and STARTJOB are synonyms; START is the preferred keyword.

<file title> is the title of the job symbolic that is to be started.

All parameters are passed by value; that is, the expressions are evaluated by the job performing the START statement, and these values are passed to the WFL compiler that is compiling and initiating the target job.

The <task id> refers to the compilation task (that is, the invocation of the WFL compiler).

After compilation of the JOBSYMBOL file, the started job is completely separated from the job that started it and runs according to its own attributes and functions.

If the started job has no usercode in the job heading, the usercode is carried forward to the started job. When the job heading is parsed, WFL runs under either the usercode of the job (if one is supplied) or the usercode of the job

that performed the START.

The <starttime spec> is passed to the invoked WFL compiler and applies to the WFL job being started.

The invoked WFL compiler determines from the <starttime spec> the absolute time and date at which the job is to begin execution. The <starttime specification> in the START statement overrides any <starttime specification> made in the <job attribute specification> list of the job that is being started.

A starttime may not be specified in the START statement for a job that is to be started at another host through BNA. Such a <starttime specification> results in an error for the job and is detected before the job is transferred to the other host.

When NAME, CHARGECODE, or BDNAME appear as <job attribute specification>s in the job being started, they can be constant string expressions that may be built by applying string parameters to the WFL job. These attributes are the the only string or name attributes that may be specified in this manner.

Example

The following job is stored in a disk file called TEST/EXAMPLE:

```
<i>BEGIN JOB EXAMPLE (REAL CLASSPARAM,
                      STRING CHARGECODEPARAM,
                      STRING FILEPARAM) NEWSOURCE=TEST/EXAMPLE;
      CLASS = CLASSPARAM;
      CHARGECODE=#CHARGECODEPARAM;
      COMPILE OBJECT/#FILEPARAM WITH ALGOL LIBRARY GO;
         COMPILER FILE CARD(KIND=DISK, TITLE=#FILEPARAM);
<i>END JOB
```

The following job excerpt starts the job TEST/EXAMPLE six times with different parameter values:

```
I:=0;
WHILE I LEQ 5 DO
BEGIN
   START TEST/EXAMPLE(30
                      "CHARGE" & STRING(I,*),
                      "TEST" & STRING(I,*);
   I:=I+1;
END
```

The following job schedules the job EX/1 to begin execution after 12:00 noon the day the job EXAMPLE is started:

```
?BEGIN JOB EXAMPLE;
    START EX/1;
       STARTTIME = 12:00;
?END JOB
```

The following job schedules the job EX/2 to begin execution after 8:00 AM the day after the job EXAMPLE is started:

```
?BEGIN JOB EXAMPLE;
    INTEGER I;
    I:=8;
    START EX/2 ("STRING PARAM", 5);
       STARTTIME = # (STRING(I,2) & ":00 ON +1");
?END JOB.
```

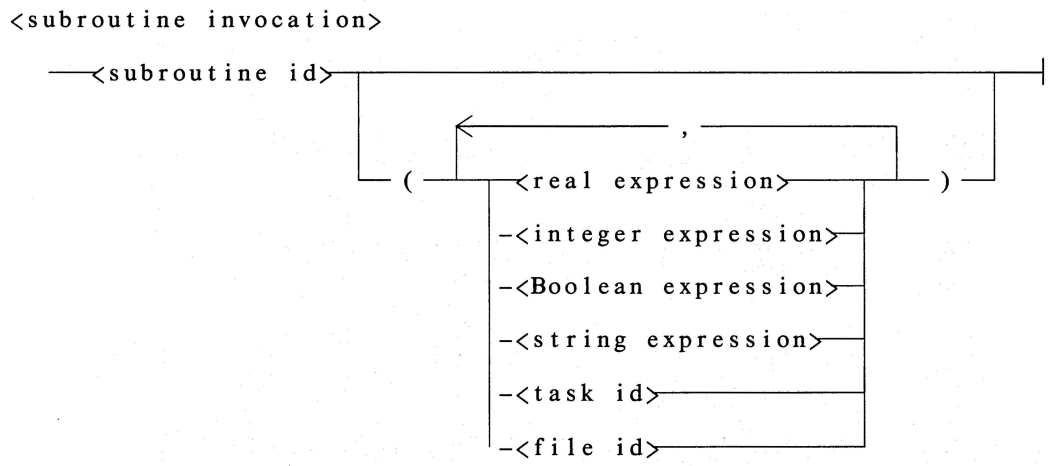
The following job schedules the job EX/3 to begin execution after 2:00 AM each of next five days after the job EXAMPLE is started:

```
?BEGIN JOB EXAMPLE;
    INTEGER I;
    I:=1;
    WHILE I LEQ 5 DO
    BEGIN
       START EX/3;
          STARTTIME = # ("2:00 ON +" & STRING(I,*));
       I:=I+1;
    END;
?END JOB.
```

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES

Subroutine Invocation Statement

Syntax



Semantics

The <subroutine invocation> statement causes a subroutine to be executed in the job stack.

The order and number of actual parameters must be the same as the order and number of the specified parameters.

If a formal parameter is specified as VALUE, then the actual parameter is passed by value; that is, a copy of the value is passed as the parameter. Modification of the parameter in the WFL subroutine changes only the value of the copy.

If a formal parameter is not specified as VALUE, then the actual parameter is passed by reference. If the actual parameter is simply an identifier, a pointer to that identifier is passed to the subroutine. Modification of the parameter in the subroutine changes the value of the identifier passed. (Task and file variables are always passed in this way). If the actual parameter is other than a simple identifier, a copy of the value is passed as the parameter. Modification of the parameter in the WFL subroutine changes only the value of the copy.

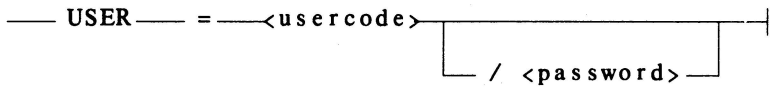
Example

The following is an example of a subroutine:

```
SUBROUTINE COMPANDGO(FILE FNAME, STRING PARAM1, REAL PARAM2);
BEGIN
     FNAME(KIND=DISK);
     COMPILE XYZ ALGOL LIBRARY;
          COMPILER FILE CARD := FNAME;
     RUN XYZ ( PARAM1, PARAM2);
     REMOVE XYZ;
     PARAM2 := PARAM2+1;
END;

REAL I;
FILE F1(TITLE=TEST1);
FILE F2(TITLE=FLOP2);

I:=1;
COMPANDGO(F1, "TEST" & STRING(I,2), I);
COMPANDGO(F2, "FLOP" & STRING(I,2), I);
```

USER Statement

Syntax



Semantics

The USER statement changes the usercode of the job. If this statement appears in a subroutine running asynchronously with the job task, it only changes the usercode of the subroutine; the usercode of the job is not changed .

The correct password must be given if such a password is connected with <usercode>. The <usercode> and <password> may only be <name constant>s.

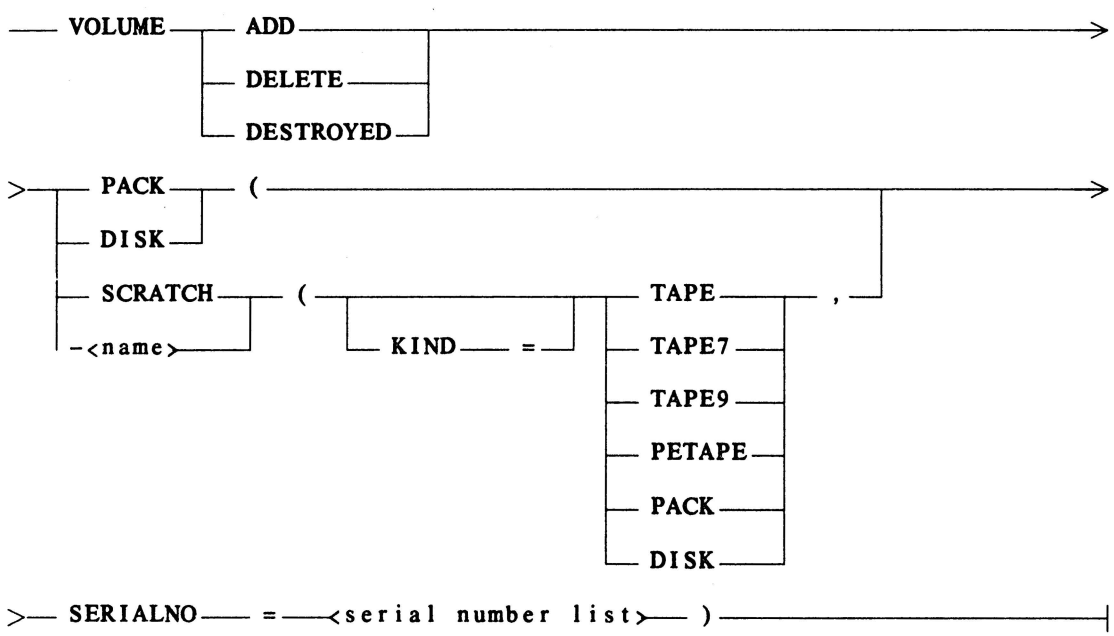
This usercode is not retained across a Halt/Load.

Examples

```
USER = MYCODE

USER = A/B
```

VOLUME Statement

Syntax



NOTE

Serial and KIND can be in any order.

Semantics

The VOLUME statement applies only to cataloged systems and is used to add a family to the volume library.

VOLUME ADD enters a volume family in the volume library. The order of the serial numbers in the statement is the order of the volumes in the family (reel number, pack index). The volume family need not be on-line. If the volume family is on-line, the information in the WFL statement is checked and expanded to include the other information stored in the entry.

A volume or volume family is deleted from the volume library by the VOLUME DELETE statement. The order in which the serial numbers appear in the statement need not be the order of the volumes in the family. A disk or pack volume family must be closed, but need not be on-line, to do a VOLUME DELETE.

The **VOLUME DESTROYED** statement is used to mark volumes that become permanently unavailable. The volume remains in the volume family and can be deleted. The action does not affect the cataloging information in the directory. To cancel the destroyed condition, the volume can be deleted and re-entered in the volume library.

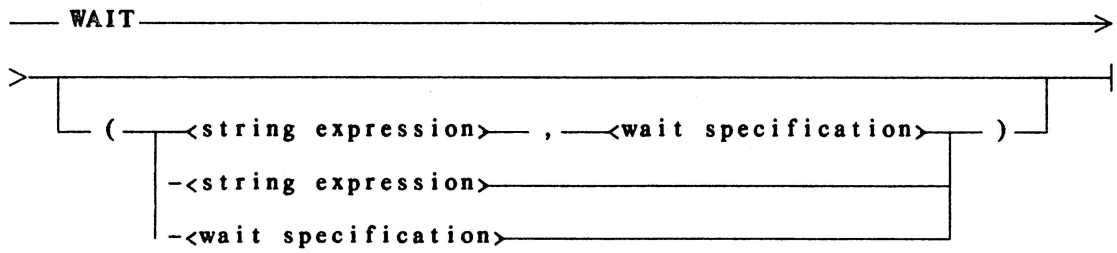
SCRATCH allows a scratch tape to be entered in the cataloged system or a volume with a particular **KIND** and **SERIALNO** to be specified as a scratch volume.

Examples

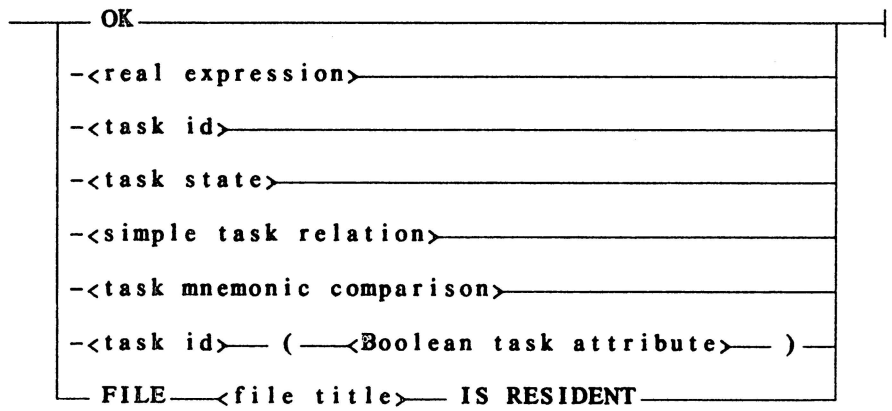
```
VOLUME ADD SCRATCH (KIND = PACK, SERIALNO = 123456)

VOLUME DELETE PACK (SERIALNO = 123456)

VOLUME DESTROYED DISK (SERIALNO = 345678)
```

WAIT Statement

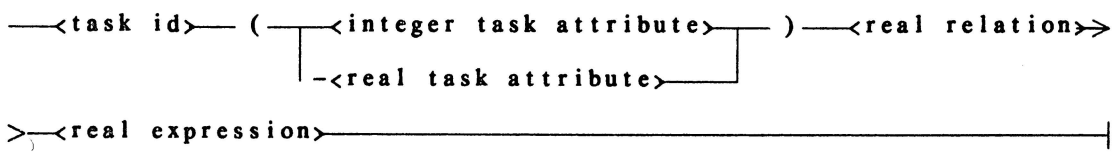
Syntax



<wait specification>



<simple task relation>



Semantics

The WAIT statement allows the job stack to suspend execution until specified conditions are met. (The string is displayed on the ODT.) These conditions are as follows:

1. The simple form causes the job task to wait for its own EXCEPTIONEVENT to be caused. The WFL job is then allowed to wait for a complex situation, such as that shown in the following example:

```
DO WAIT UNTIL
      ( T1(VALUE) = 1 OR
      T1(VALUE) = 3 OR
      T1(ACCUMPROCTIME) GTR 1000 OR
      T1 IS COMPLETED    );
```

2. WAIT (OK) - The job stack is suspended until an OK is entered from the ODT.

3. WAIT (<real expression>) - The job stack waits the specified number of seconds and then resumes execution.

4. WAIT (<task id>) - The job stack waits until the task is completed.

5. WAIT (<simple task relation>) and WAIT (<task mnemonic comparison>) - The job stack waits until either the task is completed or the task attribute satisfies the relation. The job stack waits for its own EXCEPTIONEVENT. The job does not resume execution until this event is caused and one of the above conditions is met. The EXCEPTIONEVENT of the job can be caused by the following:

   a. A task that changes task state.

   b. Entry of HI from the ODT.

   c. Programmatic cause from the task. For example, in an ALGOL task:

```
CAUSE(MYSELF.EXCEPTIONTASK.EXCEPTIONEVENT)
.
.
.
WAIT (T(VALUE) =0) ;
```

6. WAIT (<task state>) - The job stack waits until the task is completed or achieves the given state.

7. WAIT (FILE <file title> IS RESIDENT) - The job stack waits until the file is resident. This statement opens the file and displays a NO FILE on the ODT if the file is absent. The EXCEPTIONEVENT of the job need not be caused in order for this form of the WAIT statement to be completed.

NOTE

The CONTROLLER (in the MCP) removes all screen control characters found in any <string expression> that is to be displayed on the ODT.

Examples

```
WAIT (OK)

WAIT ("NEXT EVENT NOT HAPPENED", OK)

WAIT (30)

WAIT (TSK(STACKNO) GTR  T1(PROCESSTIME))

WAIT (TSK)

WAIT (TSK IS ACTIVE)

WAIT (FILE A IS RESIDENT)
```

WHILE Statement

Syntax

Semantics

The WHILE statement allows the user to perform a statement while a condition is TRUE.

The <Boolean expression> is evaluated; if the result is TRUE, the statement following the DO is executed. This sequence of events continues until the <Boolean expression> becomes FALSE or the statement following the DO transfers control outside itself.

Example

```
WHILE T(VALUE) NEQ 1 DO
  BEGIN RUN X[T];RUN Y[T];END;
```

CAUTION

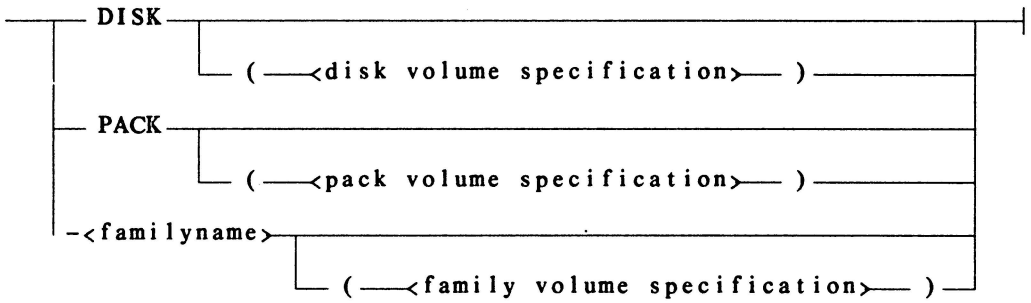
Care should be exercised to ensure that an unlimited number of RUNs are not initiated and that a <task id> is not reused before a previous occurrence terminates.

LIBRARY MAINTENANCE STATEMENTS

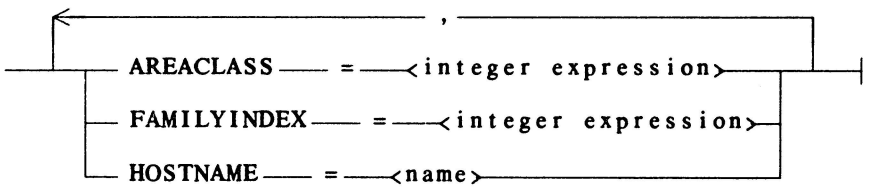
This subsection describes the following library maintenance statements: **CHANGE**, **COPY-ADD**, **REMOVE**, and **SECURITY**. The <volume specification> syntax that is common to the individual syntax diagrams of the library maintenance statements is defined below and on the next page and is not redefined with each individual statement description.

Syntax

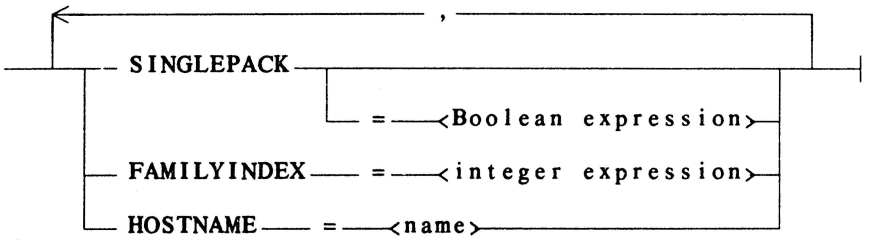
<volume specification>



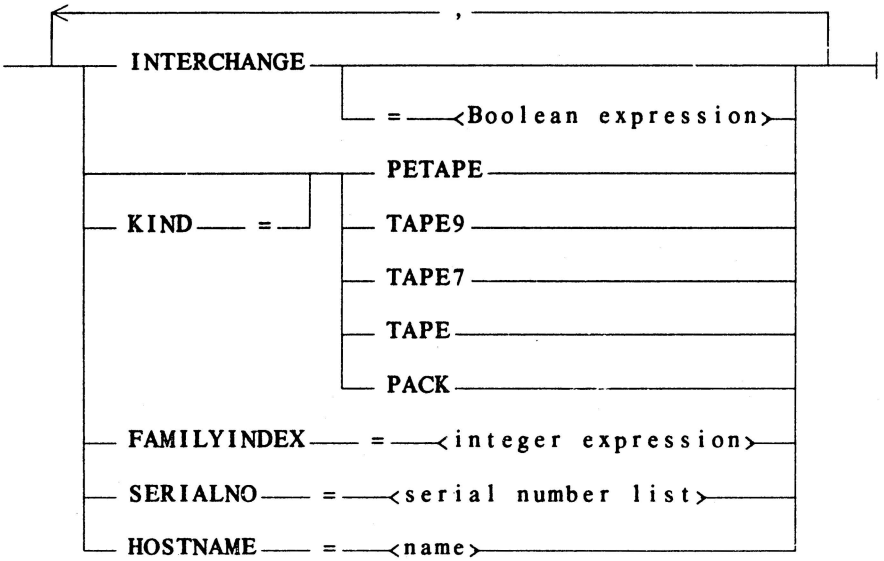
<disk volume specification>



<pack volume specification>



<family volume specification>



Pragmatics

DISKETTE is recognized as a special volume name for the library maintenance statements REMOVE and CHANGE.

Example: REMOVE X,Y,Z FROM DISKETTE(SERIALNO=3540);

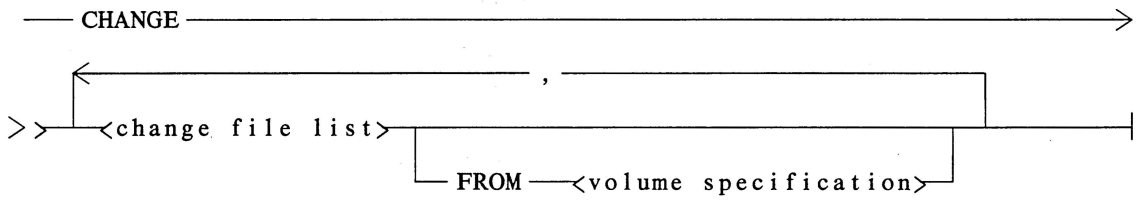
Individual DISKETTES are distinguished solely by their serial numbers. Therefore, the WFL compiler requires that a SERIALNO be specified for all DISKETTE volumes.

The COPY, ADD, CATALOG, SECURITY, and VOLUME statements do not recognize DISKETTES at the present time. If DISKETTE is specified in these statements, the statements are flagged with a NOT IMPLEMENTED YET message.

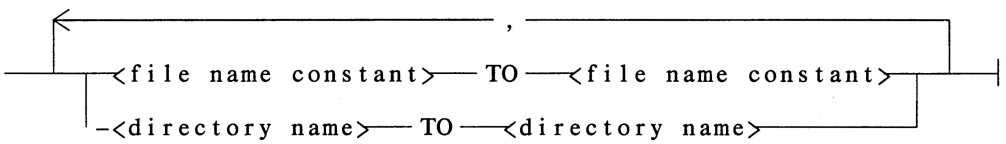
The file name must not be a multilevel name and must have no usercode prefix or * prefix. The maximum length of the file name is eight characters.

CHANGE Statement

Syntax



<change file list>



Semantics

The CHANGE statement changes the titles of files on disk or diskette.

If the <volume specification> is not specified, the files are changed on both volumes if the job has an active FAMILY statement. If the job does not have an active FAMILY statement, the files are changed on DISK. (Refer to Family Specification in Section 5.)

If a directory is specified, the titles of all files in that directory are changed.

Familynames may be used as names for individual volumes. For example:

```
CHANGE X TO Y FROM ABC(KIND=PACK);
CHANGE A TO B, C TO D FROM XYZ(KIND=PACK);
```

TEXT DELETED

CHANGE statements allow SERIALNO specifications. DISKETTE is recognized as a special volume name for the CHANGE statement. Individual DISKETTES are distinguished solely by their serial numbers. Therefore, the WFL compiler requires that a SERIALNO be specified for all DISKETTE volumes as shown in the following example:

```
CHANGE A TO B FROM DISKETTE(SERIALNO=3540);
```

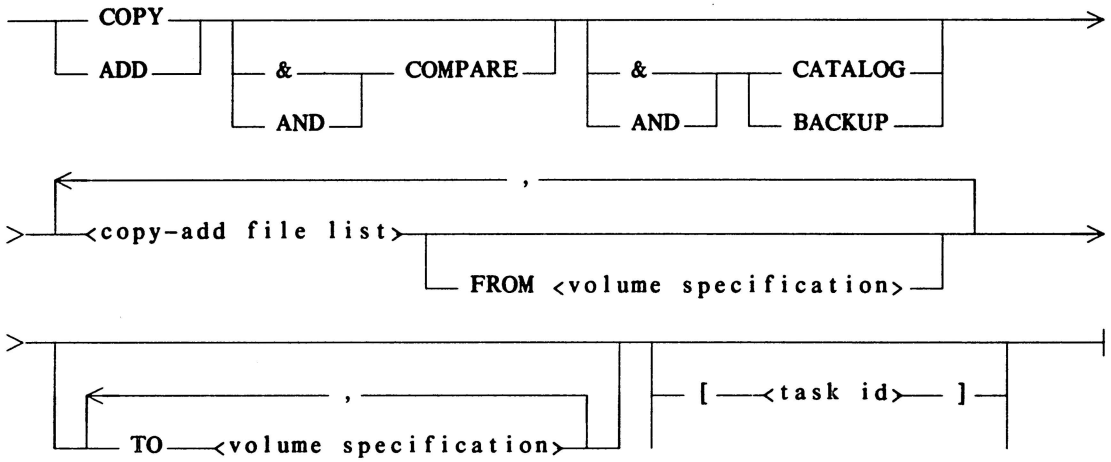
INTERCHANGE and SERIALNO attributes may be specified on DISK families.

Examples

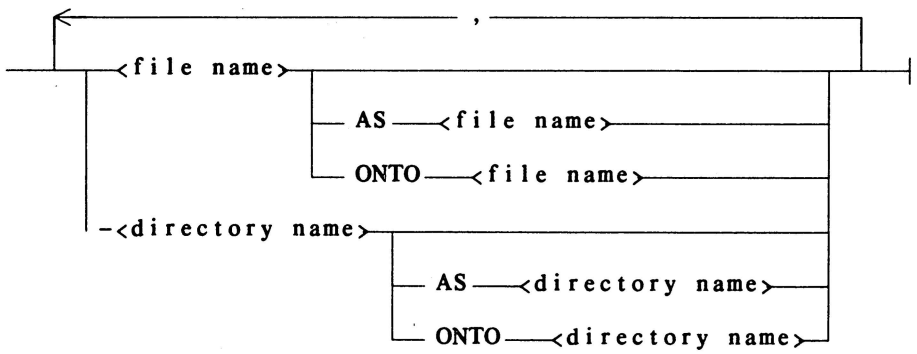
```
CHANGE X TO Y

CHANGE C/= TO D/= FROM P3

CHANGE A TO B, X/= to Y/= FROM P2

CHANGE A TO AA FROM P1, B TO BB FROM P2
```

COPY-ADD Statement

Syntax



<copy-add file list>



Semantics

The COPY statement is used to copy files to and from disk, tapes, and disk packs. If the source or destination <volume specification> does not explicitly indicate DISK or PACK, then TAPE is assumed. If an explicit TO <volume specification> is not given, the destination is assumed to be DISK. If an explicit FROM <volume specification> is not given, then the source is assumed to be DISK. The ONTO form of the <title modifier> must only be used with a destination of DISK or PACK. At least one TO or FROM <volume specification> must be given. If more than one TO <volume specification> is given, all files are copied (at the same time and in the same order) to all destination volumes. The file attribute HOSTNAME=<hostname> causes a specified file to be copied to the specified host via the COPY statement.

The AND COMPARE option causes the copied file and the original file to be compared bit for bit. If an I/O error is detected, a RECOPY REQUIRED RSVP message is displayed. The response may be an OK, OF, or DS message. The AND BACKUP and AND CATALOG are applicable only to a cataloged system. The AND BACKUP option creates a backup copy of the cataloged file. AND CATALOG allows a copied file to be entered in the cataloged system. The destination volume specification must be VOLUMEd before these files may be entered in the cataloged system.

Use of the HOSTNAME attribute is not allowed in an old WFL job.

Use of the ADD statement copies a file to a destination only if a file with the same name is not already resident on the destination. This statement is particularly useful for adding a directory to a destination where some of the files are already resident and are to be preserved. Tape is not a viable destination for an ADD statement.

The same tape volume may not be used in more than one FROM clause in the same COPY or ADD statement.

The COPY and ADD statements do not recognize the special volume name DISKETTE; if DISKETTE is specified, the statement is flagged with the message: NOT IMPLEMENTED YET.

Examples

The following examples illustrate the COPY syntax.

Copies a file X from disk to a tape Y, attaching task variable T to the copying task.

```
COPY X TO Y [T]
```

Copies file X from tape Y to disk.

```
COPY X FROM Y
```

Copies file X/Y and all files under directory Z from disk pack P to tapes T1 and T2.

```
COPY X/Y,Z/= FROM P(PACK) TO T1, TO T2
```

Copies X from disk to disk changing the name of the new copy to Y.

```
COPY X AS Y TO DISK
```

Copies X from tape T onto ("on top of") file Y on disk. Files must have the same blocking, row size, and number of rows. If the files are incompatible, a run-time error occurs.

```
COPY X ONTO Y FROM T
```

Copies everything from the system resource pack named PACK (prefixing all files by "X/") and the file A from tape B to tape T.

```
COPY = AS X/= FROM PACK, A FROM B TO T
```

Copies files A, B, and C from tape D and file E from tape F to tapes G and H.

```
COPY A,B,C FROM D,E FROM F TO G, TO H
```

Copies files X from tape Y to disk; then compares, bit for bit, file X on disk and file X on tape Y.

```
COPY & COMPARE X FROM Y
```

Copies file X from disk to tape Y; then compares, bit for bit, file X on disk and file X on tape Y. If no errors occur, file X on tape Y is entered in the directory of a catalogued system as a backup copy of the still-resident disk file X.

```
COPY & COMPARE & BACKUP X TO Y
```

Transfers file ABC from the BLUE pack on HOSTA to the RED pack on HOSTB.

```
COPY ABC FROM BLUE(KIND=PACK, HOSTNAME=HOSTA) TO RED
(KIND=PACK, HOSTNAME=HOSTB)
```

The following examples illustrate the ADD syntax.

Copies and compares file X/Y from tape T to disk if no current disk file named X/Y exists.

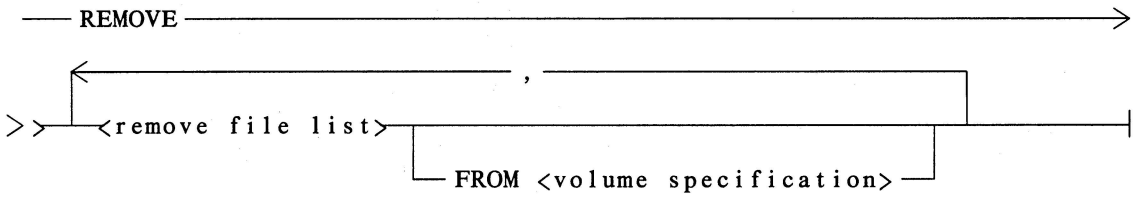
```
ADD&COMPARE X/Y FROM T
```

Loads different files to R and to disk depending on what files are present on each family.

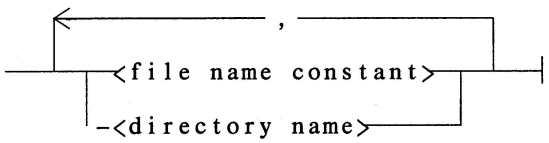
```
ADD Z/= FROM T TO R (KIND=PACK), TO DISK
```

REMOVE Statement

Syntax



<remove file list>



Semantics

The REMOVE statement removes files from disk or diskette.

If the <volume specification>is not specified, the files are removed from both volumes if the job has an active FAMILY statement. (Refer to Family Specification in Section 5 of this manual.)

Familynames may be used as names for individual volumes. For example:

```
REMOVE X FROM ABC(KIND=PACK);
REMOVE A,B,C FROM XYZ(KIND=PACK);
```

TEXT DELETED

REMOVE statements allow SERIALNO specifications. DISKETTE volumes require exactly one serial number.

Examples

The following examples illustrate the REMOVE syntax.

Removes only the file A.

```
REMOVE A
```

Removes all files in the directory A but does not remove the file A.

```
REMOVE A/=
```

Removes the three files A, B, and C from the pack family named D.

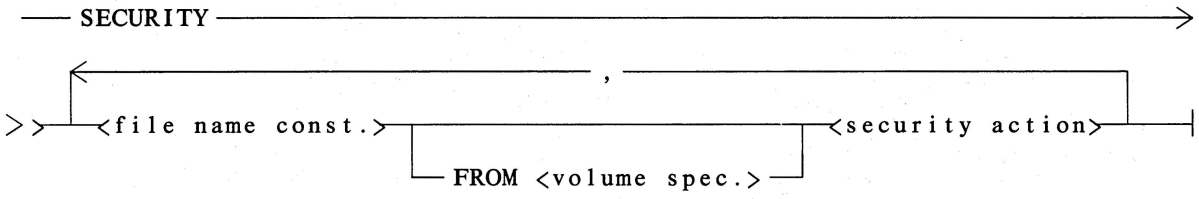
```
REMOVE A,B,C FROM D
```

Removes the files A, B, C, and D from the pack family named E and also removes the file F from the pack family named G.

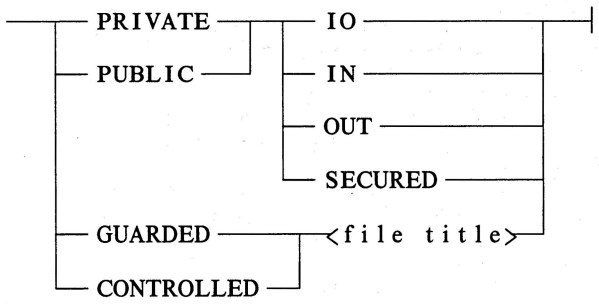
```
REMOVE A,B,C,D FROM E, F FROM G
```

SECURITY Statement

Syntax



<security action>



Semantics

The SECURITY statement is used to change the security of a file on disk. The security attributes are as follows:

1. PRIVATE — The file can be accessed only by the owner usercode of the file or by privileged users.

2. PUBLIC — Any usercode can access the file but only in accordance with SECURITYUSE privileges. (For example, IN allows read-only access; writing to the file is a security violation.)

3. GUARDED — File security is subject to the rules set forth by the file associated with <file title>, which is the associated guardfile. (Refer to SYSTEM/GUARDFILE documentation in Chapter 12 of the B 7000/B 6000 Series System Software Operational Guide, Volume 1.)

4. SECURED The file can only be executed; reads of or writes to the the file cannot be done.

5. CONTROLLED Equivalent to GUARDED, except that the guardfile is invoked even for the creating (owning) usercode. The ACCESSCODE and the optional ACCESS PASSWORD are used to control access to the file.

If the <volume spec.> is not specified, the security of the file is changed only for the primary family if the job has an active FAMILY statement. (Refer to Family Specification in Section 5 of this manual.)

Examples

The following examples illustrate the SECURITY syntax.

Changes the security of file AB/XY on DISK to PRIVATE for both input and output.

```
SECURITY AB/XY PRIVATE IO
```

Changes the security of a file named PACKB located on a pack named PACK to PUBLIC input only.

```
SECURITY PACKB FROM PACK PUBLIC IN
```

Changes the security of file FILEA located on MYPACK to be guarded by the guardfile TIOMASTER.

```
SECURITY FILEA FROM MYPACK GUARDED TIOMASTER
```

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

APPENDIX A.

TASK ATTRIBUTES AND MNEMONICS IN WFL

INTRODUCTION

This appendix describes various <task attribute>s. The attribute name is followed by a list of characteristics which contains the type of <task attribute> and other pertinent information. The type of <task attribute> must match the type required in the syntax diagrams elsewhere in this manual. For example, if <real task attribute> is required, any <task attribute> whose type is REAL may be used.

The FILEKIND attribute (along with others such as UNITS, KIND, DENSITY, and so forth) has equivalent mnemonic values. If an attribute has mnemonic values, the WFL compiler treats the attribute as a string and gives an error for any other use. If an attribute has mnemonic values, it is a MNEMONIC attribute and must be used as such. <arithmetic expression>s may not be used with mnemonic attributes. When attributes are integer in nature (that is, they return an integer to an ALGOL program), they must be used as a string in WFL.

Attributes of type MNEMONIC described in this appendix also specify compatible <task mnemonic>s. For example, ACTIVE is a compatible <task mnemonic> for the STATUS <task attribute>.

ACCESSCODE (STRING; READ/WRITE; INHERITED)

This attribute is the accesscode in use by the task for access rights to CONTROLLED files. The accesscode of the parent job is propagated to all tasks in the job unless a different accesscode is explicitly set via the <ACCESSCODE statement>. (See ACCESSCODE Statement in Section 6.)

ACCUMIOTIME (REAL; READ ONLY)

This attribute gives the accumulated I/O time for the task in seconds.

ACCUMPROCTIME (REAL; READ ONLY)

This attribute gives the accumulated processor time for the task in units of seconds.

BDNAME (STRING; READ/WRITE)

This attribute allows a prefix (other than the default prefix BD/=) to be used as the multifile ID of the backup files of the task. Any valid file TITLE may be specified as this prefix. This attribute is effective only if the task option BDBASE is ON. The construct is useful when a process other than AUTOPRINT is to print the backup files and, thus, prevent the automatic printing and removal of the files.

If a backup prefix is used by a task running under a usercode, the resulting file is placed under the directory of that user.

To set BDNAME to names of the form (C)B or *B (where C is a different usercode than that under which the task is being run), the usercode must be privileged.

CHARGECODE (B 6000=NAME; READ/WRITE; INHERITED)

This attribute contains the charge information for the task. The MCP does not use this CHARGECODE information; it merely logs the information. If a site wishes to use charging conventions, some possibilities are specified in the SYMBOL/UDSTRUCTURETABLE description. (See Section 9 of the B 5000/B 6000/B 7000 Series System Software Operational Guide, Volume 2, for a description of SYMBOL/UDSTRUCTURETABLE.)

CORE (INTEGER; READ/WRITE)

This attribute provides an estimate of memory required to schedule a task.

The first number, a data core estimate, is read/write. The second number, a code core estimate, is write-only.

DESTNAME (STRING; READ/WRITE; INHERITED)

This string attribute may be set to any station name defined in the Network Definition Language (NDL) or to the word SITE. If this attribute is set to a station name, the NDL description is interrogated, and the destination LSN and controlling MCS number are set in the path control word. If this attribute is set to SITE, the destination control portion of the path control word is set to zero.

Setting this attribute to a station name causes all printer and punch backup to be built under the directories REMLPnn or REMCPnn, respectively. The nn in the titles is the MCS number assigned to that station.

This attribute may also be read; in this case, the attribute returns the station name associated with the destination unit. If the attribute is interrogated and a remote destination has not been specified, the string SITE is returned.

The default is the station from which the job was initiated.

ELAPSEDTIME (REAL; READ ONLY)

This attribute gives the total elapsed time since the initiation of the task in seconds.

FAMILY (STRING; READ/WRITE; INHERITED)

This attribute indicates which family specifications are to be applied to a job or task. The syntax for setting a task FAMILY is described under Family Specification in Section 5 of this manual. The default FAMILY value for a JOB is obtained by the WFL compiler from the SYSTEM/USERDATAFILE entry for this usercode.

HISTORY (REAL; READ ONLY)

This attribute specifies the reason that a process is terminated. The contents of this word are primarily used to indicate why the stack is DSed or STed.

HISTORYCAUSE (MNEMONIC; READ ONLY)

This attribute indicates what caused a task to be abnormally terminated or stopped. Possible values are as follows:

OPERATORCAUSEV

PROGRAMCAUSEV

RESOURCECAUSEV

FAULTCAUSEV

SYSTEMCAUSEV

DCERRCAUSEV

IOERRCAUSEV

SOFTIOERRCAUSEV

NEWIOERRCAUSEV

UNIMPLEMENTEDCAUSEV

UNSPECIFIEDCAUSEV

EBDMSERRRCAUSEV

NETWORKCAUSEV

HISTORYTYPE (MNEMONIC; READ ONLY)

This attribute is primarily used to indicate what type of termination occurred for a task. Possible values are as follows:

NORMALV

DUMPINGV

QTEDV

STEDV

DSEDV

NORMALEOTV

SYNTAXERRORV

UNKNOWNEOTV

UNINITIATEDV

HOSTNAME (STRING; READ/WRITE; INHERITED)

This attribute determines the host system on which the task is to run or is being run. Use of this attribute indicates to the system where a resource is to be found or where an action is to take place. HOSTNAME may be read at any time but must be set prior to task-initiation time.

ITINERARY (STRING; READ ONLY; INHERITED)

This attribute contains a record of the foreign ancestory of the task. The leftmost entry in the string is the hostname of the most recent foreign ancestor of the task. The next entry in the string is the hostname of the host where the next eldest foreign ancestor is, and so forth. This attribute is inherited verbatim from parent to sibling, when the parent and offspring are running on the same host. When the parent and offspring are on different hosts, the sibling has an itinerary attribute that contains the hostname of the host where the parent is running, as the leftmost entry in the string followed by the contents of the itinerary attribute of the parent. This attribute is null until the task family ancestory first crosses a host boundary such as, a parent starting a sibling at some host other than the host where the parent is running.

The contents of the itinerary attribute for four related tasks are shown below. The relationship of the tasks is as follows: (1) task A is started on host BLUE; (2) task A starts task B on host YELLOW; (3) task B starts task C on host YELLOW; and (4) task C starts task D on host RED.

Task	Itinerary string of task
A	NULL (that is, ".")
B	BLUE
C	BLUE
D	YELLOW, BLUE

JOBNUMBER (INTEGER; READ ONLY; INHERITED)

This attribute gives the job number under which this task is being run. A task is identified to the system operator by its job number, mix number (see the STACKNO attribute), and NAME.

JOBSUMMARY (MNEMONIC; READ/WRITE)

This attribute provides a means by which the job summary can be suppressed or made available on all printers used by the job. The use of "job summary" here includes both the job summary and the WFL statements. DEFAULT is assumed if the attribute is not provided. The values are as follows:

DEFAULT	The printing of the job summary is controlled by the ODT NOSUMMARY option (option 21).
CONDITIONAL	The job summary is suppressed if no backup files are produced. The job summary is printed if a task terminates abnormally.
SUPPRESSED	No job summary is printed, even if backup files are produced. The backup files are printed with the usual beginning and ending banners.
UNCONDITIONAL	The full job summary is printed unconditionally on all printers used by the job. UNCONDITIONAL overrides the ODT NOSUMMARY option setting.

DEFAULT is assumed if the attribute value is not specified. JOBSUMMARY set to CONDITIONAL functions like OPTION = NOSUMMARY except that the NOSUMMARY bit in the OPTION word is ignored if the JOBSUMMARY attribute is set to any value. The JOBSUMMARY setting is not inherited by offspring jobs or tasks. This attribute can be read or set at anytime.

The value of this attribute is not used until EDJ. Therefore JOBSUMMARY can be set a number of times, but only the last setting before EDJ is meaningful. Because this attribute has meaning only to a job and not a task, the task designator MYJOB should be used to set JOBSUMMARY rather than MYSELF. If MYSELF is used and the task's stack is not the same as its job stack, the setting is ignored at EDJ. If the stacks are the same, using MYSELF has the same effect as using MYJOB. A program executed through a CANDE session that sets MYJOB.JOBSUMMARY results in having the setting apply to the CANDE session's job summary.

The action taken for jobs routed to a foreign host by way of BNA is dependent upon the system type and the software running at the foreign host. The JOBSUMMARY attribute is not supported by the BNA tasking facilities. Any attempt to use the attribute with an inter-host task family results in an attribute error.

Example

```
MYJOB (JOBSUMMARY = SUPPRESSED)
```

LOCKED (BOOLEAN; READ/WRITE)

The MCP does not use the value of this attribute. This attribute may be set or read at any time.

MAXCARDS (INTEGER; READ/WRITE; INHERITED)

This attribute specifies the limit of the number of cards to be punched by the task.

MAXIOTIME (REAL; READ/WRITE; INHERITED)

This attribute specifies the maximum amount of I/O time that may be used by a process. This attribute may be read or set at any time; however, the value at process initiation is saved by the system throughout process execution. Values are expressed in seconds. The specified time can be a fractional number of seconds.

MAXLINES (INTEGER; READ/WRITE; INHERITED)

This attribute specifies the maximum number of lines to be printed by the task.

MAXPROCTIME (REAL; READ/WRITE; INHERITED)

This attribute specifies the maximum amount of processor time that may be used by a process. This attribute may be read or set at any time; however, the value at process initiation is saved by the system throughout process execution. Values are expressed in seconds. The specified time can be a fractional number of seconds.

MAXWAIT (INTEGER; READ/WRITE; INHERITED)

The MAXWAIT attribute specifies the maximum number of seconds a task can wait on specified system functions, such as DMSII record locking.

NAME (TITLE; READ/WRITE)

This attribute gives the name of the task.

OPTION (MNEMONIC; READ/WRITE)

This attribute is used to set options for the process. A task may have any combination of options set at the same time. The syntax for setting a task option is described under Task Attribute Assignment in Chapter 5.

The options, when set, have the following meanings:

ARRAYS	All arrays of the stack are dumped if and when PROGRAMDUMP occurs.
AUTORM	Duplicate files for this task are automatically removed.
BACKUP	Regardless of the BDONLY setting, printer files are sent to backup disk.
BASE	The base of the stack is dumped if and when PROGRAMDUMP occurs.
BDBASE	The descendent stack is initiated as a separate job rather than as a separate task.
CODE	The Segment Dictionary of the task is dumped if and when PROGRAMDUMP occurs.
DBS	The database SIB and the database stack are dumped if and when a PROGRAMDUMP occurs.
DEBUG	COBOL74 or FORTRAN programs execute special compiled-in debugging code (refer to the B 5000/B 6000/B 7000 Series COBOL ANSI-74 Reference Manual and the B 7000/B 6000 Series FORTRAN Reference Manual).
DSED	Program dump occurs for any external termination condition (such as operator DS).
FAULT	Program dump occurs if any fatal internal error occurs.
FILE	FILE is a synonym for FILES.
FILES	Files of the stack are dumped if and when PROGRAMDUMP occurs.
LIBRARIES	All libraries associated with the stack are dumped if and when PROGRAMDUMP occurs.
LONG	No arrays are segmented.
NOSUMMARY	No JOB SUMMARY is printed if no BD files are present. The JOB SUMMARY is printed if a task terminates abnormally.

PRIVATELIBRARIES
Libraries private for that stack are dumped if and when PROGRAMDUMP occurs.

PRIORITY (INTEGER; READ/WRITE; INHERITED)

This attribute is used by the system for scheduling. PRIORITY informs the system of the urgency with which the task should be completed. This attribute is used by the system to determine (1) the order of selection of active processes from the schedule and (2) the order of assignment of processes to the processor(s). Consequently, the order for allocation of primary storage and the order of locking resources and peripherals are also determined by this attribute. PRIORITY may be used to override the default priority.

RESTART (INTEGER; READ/WRITE)

The value of this attribute specifies the number of times the task is to be re-executed following an error termination and pending a successful termination. RESTART is decremented by one after each execution of the process. An operator-DSed task does not restart.

RESTARTED (BOOLEAN; READ ONLY)

This attribute is set to TRUE if the task has been restarted for any reason; the attribute is FALSE otherwise.

STACK (INTEGER; READ/WRITE)

This attribute specifies or returns the stack size requirement used for scheduling by the system. This attribute may be read or set at any time; however, the value at process initiation is saved throughout process execution. The maximum value is 16384.

STACKLIMIT (INTEGER; READ/WRITE)

This attribute sets a limit on how far a stack can be stretched. If this limit is exceeded, a STACKOVERFLOW condition occurs.

STACKNO (INTEGER; READ ONLY)

This attribute returns the mix number of an active process or the negative of the mix number of a terminated process. This attribute may be read at any time. A negative number indicates a terminated process, a positive number indicates an active process, and a zero indicates that the task variable was never associated with an active process.

STATION (INTEGER; READ/WRITE; INHERITED)

This attribute, used for Data Comm only, contains the number of the station from which the task was initiated. The attribute may be read or written, but the value at process initiation is saved by the system for use throughout process execution.

STATUS (MNEMONIC; READ/WRITE)

This attribute returns the current state of the process. This attribute may be set to cause status changes as described below and may be read or set at any time.

NEVERUSED	Not being used (the task is not associated with a process). Setting the attribute to NEVERUSED disassociates the task from the process.
SCHEDULED	The process is scheduled. Setting the attribute causes the process to be scheduled.
ACTIVE	The process is active. Setting the attribute to ACTIVE has no effect unless the process is suspended.
SUSPENDED	The process is suspended. Setting the attribute to SUSPENDED suspends the process.
TERMINATED	The process is terminated (DSed or EOT). Setting the attribute to TERMINATED causes the process to be DSed.
BADINITIATE	Initiation of the process failed when attempted by the MCP procedure DOCTOR. Setting this attribute to BADINITIATE has no effect. The DOCTOR procedure stores an error code in task, and HISTORY indicates the cause of the failure. (Refer to the HISTORY attribute in this appendix.)

The STATUS task attribute may not be used in the task equation list of a task initiation.

The following example is not valid:

```
?BEGIN JOB TEST;
  TASK T;
  RUN X[T];
    STATUS=NEVERUSED;          % syntax error
?END JOB
```

SUBSPACES (INTEGER; READ/WRITE; INHERITED)

The values (0 thru 3) of this attribute determine the manner in which the SWAPPER independent runner handles the associated tasks. The four values are:

0	The code, stack, and data of the task are not contained in the swap area.
1	The code of the task is not contained in the swap area.

2 If the code file of the task is in the usercode library, the MCP places the code, stack, and data of the stack in the swap area. In all other cases, the code is placed outside the swap area, and the stack and data are placed in the swap area.

3 The code, stack, and data of the task are contained in the swap area.

SUBSYSTEM (STRING; READ/WRITE)

This attribute specifies the subsystem on which the task is to run or is running. This attribute may be set only where the task is inactive and is not automatically propagated from a parent task to descendent tasks. If set, this attribute is treated only as a request, not as a binding demand to guide the placement of a task. In one of the following three circumstances, the attribute setting may be disregarded:

1. The name subsystem is undefined.

2. The name subsystem is empty.

3. The task was constrained by an ancestral relationship to a subsystem outside the named set. The firm rule is that a dependent task must not be more global than the parent of the task.

SW1 THROUGH SW8 (BOOLEAN; READ/WRITE; INHERITED)

These eight attributes may be set or tested for user-specified purposes as desired.

TANKING (MNEMONIC; READ/WRITE)

This attribute allows specification the default value for tanking on a task basis for all remote files in the task. When tanking is specified, information is temporarily stored on disk when the queue is full. This information is retrieved and placed in the queue as the queue empties. The values are as follows:

UNSPECIFIED File is not tanked unless changed by the MCP at file-assignment time. This value is the default state.

NONE File is not tanked, and an MCS is prevented from specifying TANKING.

SYNC File is tanked. When the file is closed, the task does not continue until all tanked output has been completed.

ASYNC File is tanked. The task continues past the file close and does not wait until all tanked output has been completed. This specification allows a task to go to end-of-task (EOT) while its output continues to be sent.

TASKVALUE (REAL; READ/WRITE)

This attribute may be set or tested for specific purposes as desired. Values are user-specified. This task attribute is used as a means of communication between processes and is known to WFL as a value.

USERCODE (STRING; READ/WRITE; INHERITED)

This attribute specifies the usercode under which the task is run. The usercode of the parent job is propagated to all tasks in the job.

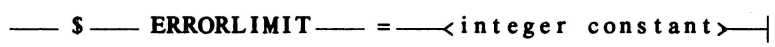
APPENDIX B. WFL CONTROL OPTIONS

INTRODUCTION

WFL control options can be used to affect the manner in which WFL processes source card images. The options appear on card images with a dollar sign in column 1 or 2. The dollar sign may be followed by one or more of the WFL control options. If the dollar sign is in column 2 and NEWSOURCE is specified, that card image is written to the new source file; otherwise, it is not written.

ERRORLIMIT WFL CONTROL OPTION

Syntax



Semantics

Compilation of the job is terminated if the number of errors detected by the WFL compiler becomes greater than or equal to the <integer constant>. If no $ERRORLIMIT card appears, the default error limit is 9999 unless the job was started through CANDE. If the job was started through CANDE, the default limit is 10.

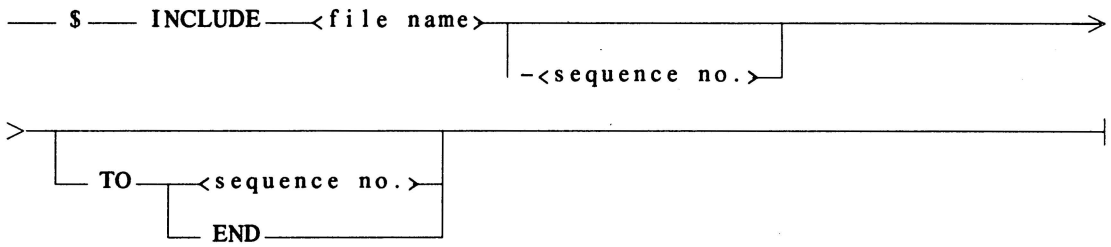
Example

In the following example the number of errors detected during compilation of the job cannot be greater than or equal to 50.

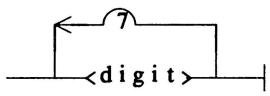
```
$ ERRORLIMIT = 50
```

INCLUDE WFL CONTROL OPTION

Syntax



<sequence no.>



Semantics

The INCLUDE WFL control option includes symbolic text during compilation. <file name> is the name of a file (FILETYPE=JOBSYMBOL) which contains the text to be included. An INCLUDE may appear in a START file or in a WFL batch job deck. The text is included each time the file is STARTed.

The $ may appear in column 1 or 2 of the card image. If NEWSOURCE is specified and if the $ appears in column 1, the INCLUDEd text is written to the new source file in place of the INCLUDE card image. If the $ appears in column 2, the INCLUDE card image is written to the new source file. The INCLUDEd text is listed in the control-card listing of the job summary printout.

A SERIALNO is required when a minidisk is specified in an INCLUDE option.

The optional sequence range specifies the lines of the INCLUDEd file. If the sequence range is omitted, the entire file is included.

INCLUDE options may not be nested.

After parsing the <job attribute specification> WFL runs under the USERCODE and FAMILY of the job (if they are supplied). Therefore, if an INCLUDE is encountered after the job heading is parsed, the directory of the usercode is searched for the file to be INCLUDEd.

Pragmatics

If an INCLUDE is encountered within the heading of a WFL job, one of the following conditions must be true:

1. The security of the file being INCLUDEd must be PUBLIC IO or PUBLIC IN.

2. The file being INCLUDEd must have the same usercode as the disk job symbolic file that contains the $INCLUDE record.

These two restrictions still allow jobs entered through a card reader or STARTed from the ODT to INCLUDE any PUBLIC file. STARTed and ZIPped jobs also allow job heading information to be INCLUDEd from other files with the same usercode as that with which the job symbolic itself is stored.

Example

```
<i>BEGIN JOB ZZZ;
RUN SYSTEM/CARDLINE;
$INCLUDE CARDLINE/ATTRIBUTES ON MYPACK
      :
      :
<i>END JOB
```

NOTE

<file name> may not be a string variable (#s).

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

INDEX

Documentation Evaluation Form

Title: B 5000/B 6000/B 7000 Series Work Flow Language Reference Manual

Form No: 5011794

Date: March 1981

Burroughs Corporation is interested in receiving your comments and suggestions regarding this manual. Comments will be utilized in ensuing revisions to improve this manual.

Please check type of Suggestion:

☐ Addition ☐ Deletion ☐ Revision ☐ Error

Comments:

From:

Name _____

Title _____

Company _____

Address _____

_____

Phone Number _____ Date _____

Remove form and mail to:

Burroughs Corporation
Documentation Dept., TIO - West
P.O. Box 4040
El Monte, CA 91734
U.S.A.