Constrained Materials Management and Production Planning Tool

Release 8.0

User’s Guide and Reference

Revised August 29, 2019

Fourth Edition (August 2011)

PREFACE

About this Book

Constrained Material Management and Production Planning Tool (WIT) is a software tool for constrained materials and production planning. It performs an implosion or explosion on a set of parts, product demands, and multilevel bill-of-manufacturing (BOM). WIT consists of a stand-alone executable which reads and produces flat files and an Application Program Interface (API) that provides communication through function calls.

Who Should Use This Book

This book is written for WIT users. There are some sections written predominately for application programmers. It assumes the programmers are familiar with the C programming language. This book assumes all readers are familiar with general manufacturing concepts. For a detailed introduction to WIT, see [1].

How this Book is Organized

Chapter 1–Introducing WIT

- defines terminology and provides an overview of WIT data
- briefly describes the two alternative methods used to solve the implosion problem
- gives some properties of WIT's implosion heuristic
- defines the Objective functions for WIT's optimization
- describes the additional capabilities of WIT

Chapter 2–WIT Data

WIT has seven major kinds of data objects. They are: the WIT problem itself, parts, demands, operations, BOM entries, substitute BOM entries, and BOP entries. Each object has a list of attributes which fully describes the object. Chapter 2 describes data objects and their attributes and contains an alphabetical index of data attributes.

Chapter 3–Using the Stand-alone Executable

This chapter contains a description of the WIT Stand-alone Executable and how it is used.

Chapter 4–Using the API: Application Program Interface

This chapter contains descriptions of WIT data types used by the API including bound sets and message formats. A table describing WIT data attributes in terms of how they are used by API functions is included.

Chapter 5–Function Library

The Function Library contains all the WIT functions in alphabetical order by type.

Appendix A–API Sample Code

Appendix A contains two samples of API code listings.

Appendix B–File Formats

Appendix B contains file formats for input and output files:

- Input Data file
- Control Parameter file
- Execution Schedule file
- Shipment Schedule file
- WIT-MRP Requirements Schedule file
- Critical Parts List output file

References

Index

General Information about this Book

The following sections explain conventions and the other information you need to make using this book easier.

Finding and Interpreting a Function Description

All API functions are described by type in alphabetic order by the type in Chapter 5 “API Function Library”. Functions are listed individually under API Functions in the Index. You can also refer to the Table of Contents.

For each function, a description of what the function does is followed by definitions of its parameters. Other programming points and restrictions to consider appear under the headings Usage Notes and Error Conditions. A coding example of the function and a short explanation are found at the end of each function description.

Interpreting the Fonts

A non-proportional font is used to distinguish code from the normal text in this book.

Font Examples
Normal text uses this proportional font.
`/* Code uses this non-proportional font. */` `witAddPart( theWitRun, “PartA”, WitMATERIAL );`

Abbreviations

Abbreviations used in this book are defined below.

Short Name	Full Name
API	WIT’s Application Program Interface
BOM	Bill-of-Manufacturing
BOP	Bill-of-Products
E/C	Engineering Change
FSS	Focussed Shortage Schedule
LP	Linear Programming
MRP	Material Requirements Planning
WIT	Constrained Materials Management and Production Planning Tool
WIT-MRP	WIT’s MRP feature

Special Symbols

This book uses the following graphic and mathematical symbols:

☞ The symbol shown here appears in the left margin when an example is given.

The symbol shown here appears in the left margin to bring attention to a note that provides specific information.

The meaning of the mathematical symbols used in this book are as follows:

$\forall$ means “for all”

$\in$ means “elements of”

$\sum$ means “sum”

$+\infty$ means “positive infinity”

Notices

Trademarks and Service Marks

The following terms, denoted by an asterisk (*) in this book, are trademarks for the IBM Corporation in the United States and/or other countries:

- AIX
- IBM
- RISC System/6000 (RS/6000)

Default File Names for the Stand-Alone Executable on AIX

As explained in Chapter 3 “Using the WIT Stand-Alone Executable”, the Stand-Alone Executable communicates primarily through flat disk files. The Table below lists the default file names when the WIT Stand-Alone Executable is running on AIX*.

The Memo of Licensees and the README file will show the default file names for the licensed program.

TABLE 1 Default File Names Under AIX (Stand-Alone Executable)

File	Default Name
Control Parameter Input File	run.params
Input Data File	wit.data
Status Log File	log.out
Echo Output File	echo.out
Pre-processing Output File	pre.out
LP Solver Log File	solver.log
Comprehensive Implosion Solution Output File	soln.out
Comprehensive MRP Solution Output File	soln.out
Execution Schedule Output File	exec.out
Shipment Schedule Output File	ship.out
Requirements Schedule Output File	mrpsup.out
Critical Parts List Output File	critical.out

API Compiling, Linking, and Running

The following commands are shown as a general reference. The Makefile file found in the samples directory shows an example of compiling and linking the three sample programs.

Some hints for running your program are:

- Check that the printing of useful messages has not been turned off with `witSetMesgTimesPrint`.
- Try running a simplified version of your program to isolate any errors.
- The function `witWriteData` can be used to used to write out the current contents of the WIT problem. This format of the model can be used to capture the model data when reporting problems.
- If the program runs out of space when solving large problems, see "Temporary Files" on page viii and "Getting Enough Memory" on page ix.

Temporary Files

When running your program WIT will create temporary files. Error message WIT0418S may indicate a lack of disk space

When using AIX, the environment variable `TMPDIR` can be specified to indicate the directory where the temporary files are to be placed. If `TMPDIR` is not specified then `/tmp` is used. If you are running out of disk storage when running WIT, try setting `TMPDIR` to the name of a directory that has more free space.

If the WIT application prematurely ends, temporary files may not be deleted. All WIT temporary files are prefixed with the characters `wit`. These files may need to be removed by some other means.

Getting Enough Memory

When solving large problems, especially with optimization, WIT requires large amounts of main memory or storage. Your system may have default limits on the amount of memory you can use, resulting in error messages such as WIT0103S or Segmentation fault.

If you are using AIX, the following may help. Make sure that the maximum amount of “physical” or “real” memory and the maximum data segment size are either very large or unlimited. (Unlimited is better). (If you are using the C shell (csh), you can check this using the `limit` command. `Data size` and `memory use` should be very large or unlimited.) If your real memory or segment size are limited, do the following:

1. As superuser (root), enter the command "smit user"
2. Use the "Change/Show Characteristics of a User" option.
3. Enter your User-Name/login-id.
4. Change “Max physical MEMORY” to 0 (for unlimited) or some large number.
5. Change “Max DATA Segment”. SMIT will not allow you to set it to 0. That will be done in the next step.
6. As root, edit /etc/security/limits. Replace the number after "`data =`" in the stanza for your login-id to 0 (for unlimited).
7. Log out and log back in so that the changes take effect.

Your system administrator may have to do this for you.

Installed Files and Directories

The following is a general list of files and directories. Since this information is platform specific, please refer to the Memo to Licensees or the README file for detailed information.

• lib/libwit.a	WIT API library.
• include/wit.h	WIT header file used when compiling WIT applications.
• bin/wit	WIT stand-alone executable.
• samples/sample1.c	API sample program.
• samples/sample2.c	API sample program.
• samples/sample3.c	API sample program.
• samples/sample.data	Sample Input Data file used by the WIT stand-alone executable and sample2.
• samples/run.params	Sample run.params file used by the WIT stand-alone executable.
• samples/Makefile	Makefile for the sample programs.

Summary of WIT Software Changes

Change History Release 6.0

1. The input data file format now accepts “6.0” in the release specification.
2. A new optional capability has been added to heuristic implosion and allocation, called “stock reallocation”. This capability allows the heuristic to consume stock of a part in one period that was previously allocated to be consumed in a later period and then to produce the part in the later period, in order to make up for the consumed stock. This capability is controlled by a new global attribute, “stockRealloc”. The new API functions are:
   - `witSetStockRealloc`
   - `witGetStockRealloc`
3. The “build-ahead by demand” capability of heuristic implosion and allocation is no longer documented in this guide. For upward compatibility, the API functions associated with this capability continue to function correctly; they are just not documented. (In general, the NSTN and ASAP build-ahead capabilities can be used to create the same effect.) The affected API functions are:
   - `witSetDemandBuildAheadUB`
   - `witGetDemandBuildAheadUB`
   - `witSetDemandPrefBuildAhead`
   - `witGetDemandPrefBuildAhead`
4. The rules for determining the multi-exec direction in two-way multiple execution periods have been simplified.
5. The following three scalar attributes have been replaced by vector attributes:

Object Type	Old Scalar Attribute	New Vector Attribute
BOM Entry	usageRate	consRate
Substitute	usageRate	consRate
BOP Entry	prodRate	productRate

   The new API functions are:
   - `witSetBomEntryConsRate`
   - `witGetBomEntryConsRate`
   - `witSetSubsBomEntryConsRate`
   - `witGetSubsBomEntryConsRate`
   - `witSetBopEntryProductRate`
   - `witGetBopEntryProductRate`

   The following API functions are no longer documented in this guide:

- `witSetBomEntryUsageRate`
- `witGetBomEntryUsageRate`
- `witSetSubsBomEntryUsageRate`
- `witGetSubsBomEntryUsageRate`
- `witSetBopEntryProdRate`
- `witGetBopEntryProdRate`

However, for upward compatibility, these functions still exist and operate either by setting the vector attribute to the given scalar value in all periods, or by retrieving the period 0 value of the vector attribute.

6. Heuristic implosion and allocation now have the optional ability to respect soft lower bounds on stock volumes. This capability is controlled by two new global boolean attributes: respectStockSLBs and prefHighStockSLBs. The new API functions are:

- `witSetRespectStockSLBs`
- `witGetRespectStockSLBs`
- `witSetPrefHighStockSLBs`
- `witGetPrefHighStockSLBs`

7. A new optional capability has been added to heuristic implosion and allocation, called “pegging”. This feature keeps track of the association between the resources that are being allocated by heuristic implosion and allocation and the demands for which they are being allocated and then provides this information to the application program. This feature is controlled by a new global input attribute, “perfPegging”. The new API functions are:

- `witSetPerfPegging`
- `witGetPerfPegging`
- `witGetDemandExecVolPegging`
- `witGetDemandSubVolPegging`
- `witClearPegging`

8. At the user’s request, optimizing implosion now computes shadow prices for the supplyVols of parts. There are 2 new attributes for this:

- problem.compPrices
- part.shadowPrice

And there are 3 new API functions:

- `witSetCompPrices`
- `witGetCompPrices`
- `witGetPartShadowPrice`

9. The stock reallocation optional feature of heuristic allocation and implosion has been enhanced. It now recognizes cases in which, due to resource constraints, stock reallocation needs to be done in periods other than the last possible period or not at all, and it takes appropriate action for such cases.

This new version of stock reallocation is controlled by a new global attribute, “stockReallocation”. The new API functions are:

- `witSetStockReallocation`
- `witGetStockReallocation`

There is no longer any need to do the older form of stock reallocation, so it is no longer documented in this guide. However, in order to maintain upward compatibility, the old global variable, “stockRealloc” (which controls the old form of stock reallocation) still exists and functions as before, but is no longer documented in this guide. Similarly the following API functions still operate as before, but are no longer documented in this guide:

- `witSetStockRealloc`
- `witGetStockRealloc`

Finally, having “stockRealloc” and “stockReallocation” both TRUE at the same time results in a severe error.

10. A new optional capability has been added to heuristic implosion and allocation, called “selection splitting”. This capability enhances NSTN and ASAP build-ahead, by allowing them to use more than one build period in order to satisfy requirements on a part in a single period, corresponding to one unit of top-level demand. In similar way, it also enhances:

- Multiple Routes
- Multiple Execution Periods
- Stock Reallocation
- Penalized Execution

This capability is controlled by a new global attribute, “selSplit”. The new API functions are:

- `witSetSelSplit`
- `witGetSelSplit`

11. A new data management capability has been added, called “object iteration”. The object iteration facility is a collection of API functions that enable the application program to traverse all of the data objects of a WIT problem in the order in which they were created. The object iteration facility includes one new global output attribute, “objItrState”, and nine new API functions:

- `witAdvanceObjItr`
- `witGetObjItrState`
- `witGetObjItrPart`
- `witGetObjItrDemand`
- `witGetObjItrOperation`
- `witGetObjItrBomEntry`
- `witGetObjItrSubsBomEntry`
- `witGetObjItrBopEntry`
- `witResetObjItr`

12. A new optional capability has been added to heuristic implosion and allocation, called “single-source”. This capability is an enhancement to the multi-route feature. When single-source is requested for a part, the heuristic will initially attempt to ship the entire desIncVol specified by witIncHeurAlloc (or its equivalent in heuristic implosion) by using only one BOP entry to fill the requirements of that part. Similar logic applies to the substitutes associated with a BOM entry for which single-source is requested.

There are 2 new attributes for this capability:

- part.singleSource
- bomEntry.singleSource

And there are 4 new API functions for it:

- `witSetPartSingleSource`
- `witGetPartSingleSource`
- `witSetBomEntrySingleSource`
- `witGetBomEntrySingleSource`

13. A new optional capability has been added to heuristic implosion and WIT-MRP, called “two-level lot sizes”. When this feature is used, an operation applies different minLotSize and incLotSize attributes for large execVols than it does for small execVols.

There are 4 new attributes for this capability:

- operation.twoLevelLotSizes
- operation.lotSize2Thresh
- operation.minLotSize2
- operation.incLotSize2

And there are 8 new API functions for it:

- `witSetOperationTwoLevelLotSizes`
- `witGetOperationTwoLevelLotSizes`
- `witSetOperationLotSize2Thresh`
- `witGetOperationLotSize2Thresh`
- `witSetOperationMinLotSize2`
- `witGetOperationMinLotSize2`
- `witSetOperationIncLotSize2`
- `witGetOperationIncLotSize2`

14. A new optional capability has been added to heuristic implosion and allocation, called “user-specified heuristic starting solution”. This feature allows the user or application program to specify a starting solution for heuristic implosion/allocation.

There is one new attribute for this capability:

- problem.userHeurStart

And there are 2 new API functions for it:

- `witSetUserHeurStart`
- `witGetUserHeurStart`

15. The penalized execution feature has been extended to include two new penalty attributes:

- bomEntry.execPenalty
- subEntry.execPenalty

And there are 4 new API functions:

- `witSetBomEntryExecPenalty`
- `witGetBomEntryExecPenalty`
- `witSetSubsBomEntryExecPenalty`
- `witGetSubsBomEntryExecPenalty`

16. A new API function has been added that terminates heuristic allocation without post-processing. The new API function is:

- `witShutDownHeurAlloc`

17. A new optional capability has been added to heuristic implosion and allocation, called “proportionate routing”. The proportionate routing technique applies to the same cases as the multiple routes technique. However, instead of using one BOP entry at a time to produce a given part, the proportionate routing technique uses all of the BOP entries for the part at the same time, initially, according to fixed ratios specified by the user. If a BOP entry runs into a constraint that prevents it from producing more of the part, that BOP entry is allowed to drop out and the proportionate routing is applied only to the BOP entries that remain active. The same technique applies to the use of substitutes for a BOM entry.

There are 5 new attributes for this capability:

- part.propRouting
- bomEntry.propRouting
- bopEntry.routingShare
- bomEntry.routingShare
- subEntry.routingShare

And there are 10 new API functions for it:

- `witSetPartPropRouting`
- `witGetPartPropRouting`
- `witSetBomEntryPropRouting`
- `witGetBomEntryPropRouting`
- `witSetBopEntryRoutingShare`
- `witGetBopEntryRoutingShare`
- `witSetBomEntryRoutingShare`
- `witGetBomEntryRoutingShare`
- `witSetSubsBomEntryRoutingShare`

- `witGetSubsBomEntryRoutingShare`

18. All values in WIT that are nominally of type “float” or “vector of floats” are now stored internally as type “double” or “vector of doubles.”

19. A group of new API functions has been added that have double precision arguments. Specifically, for each API function that has at least one argument whose type involves “float”, there is now a second API function whose corresponding argument(s) involve “double”. In each case, the “double” version of the function has the same name as the “float” version, but with the letters “Dbl” appended to the end of the name. The two functions perform the same essential task, but the float version must do type conversion between the float arguments and the double internal values, while the double precision version does no type conversion.

Example:

- `witSetPartSupplyVolDbl`

Number of new API functions: 136.

20. A new global boolean attribute has been added: “highPrecisionWD”. If it is set to TRUE, much more precision is used by `witWriteData` when it writes out a float value. The new API functions are:

- `witSetHighPrecisionWD`
- `witGetHighPrecisionWD`

21. A new version of pegging has been added, called “post-implosion pegging” (PIP). It is similar to the earlier form of pegging (now called “concurrent pegging”), but differs in the following ways:

- It applies to any implosion solution: heuristic, optimizing, or user-specified.
- It is not a reflection of the way in which the solution was constructed.
- The partial solution that’s pegged to any shipment is (in a certain sense) a feasible solution in its own right.
- More attributes are pegged.

The following attributes were added for this feature:

- problem.pipSeqFromHeur (input)
- problem.pipExists (output)
- bopEntry.pipShare (input)

The following API functions were added for this feature:

- `witClearPipSeq`
- `witAppendToPipSeq`
- `witSetPipSeqFromHeur`
- `witGetPipSeqFromHeur`
- `witGetPpipSeq`
- `witSetBopEntryPipShare`
- `witSetBopEntryPipShareDbl`
- `witGetBopEntryPipShare`
- `witGetBopEntryPipShareDbl`

- `witBuildPip`
- `witGetPipExists`
- `witGetDemandExecVolPip`
- `witGetDemandExecVolPipDbl`
- `witGetDemandSubVolPip`
- `witGetDemandSubVolPipDbl`
- `witGetDemandSupplyVolPip`
- `witGetDemandSupplyVolPipDbl`
- `witGetDemandProdVolPip`
- `witGetDemandProdVolPipDbl`
- `witGetDemandConsVolPip`
- `witGetDemandConsVolPipDbl`
- `witGetDemandSideVolPip`
- `witGetDemandSideVolPipDbl`

22. When penalized execution and proportionate routing are both used in the same problem, they now interact in either of two distinct modes:

- Overriding proportionate routing, in which proportionate routing overrides penalized execution, or
- Tie breaking proportionate routing, in which proportionate routing breaks ties in penalized execution.

There is one new attribute for this capability:

- problem.tieBreakPropRt

And there are 2 new API functions for it:

- `witSetTieBreakPropRt`
- `witGetTieBreakPropRt`

23. A new capability has been added to heuristic implosion and allocation, called “pegged critical list”. This is an ordered list in which each element consists of a critical part, a critical period, a corresponding demand, and a corresponding shipment period.

There are two new attributes for this capability:

- problem.pgdCritListMode
- problem.pgdCritList

And there are three new API functions for it:

- `witSetPgdCritListMode`
- `witGetPgdCritListMode`
- `witGetPgdCritList`

24. The boolean scalar “propRouting” attribute on parts and BOM entries has been replaced by the boolean vector “propRtg” attribute on parts and BOM entries.

The new API functions are:

- `witSetPartPropRtg`

- `witGetPartPropRtg`
- `witSetBomEntryPropRtg`
- `witGetBomEntryPropRtg`

The following API functions are no longer documented in this guide:

- `witSetPartPropRouting`
- `witGetPartPropRouting`
- `witSetBomEntryPropRouting`
- `witGetBomEntryPropRouting`

However, for upward compatibility, these functions still exist and operate either by setting the vector attribute to the given scalar value in all periods, or by retrieving the period 0 value of the vector attribute.

25. Attributes for objective #1 can now be set and retrieved even when objChoice != 1. This applies to the following attributes:

- part.obj1ScrapCost
- part.obj1StockCost
- opn.obj1ExecCost
- subEntry.obj1SubCost
- demand.obj1ShipReward
- demand.obj1CumShipReward

The objChoice attribute can now be set even when parts and/or operations exist.

26. The default value of the global objChoice attribute has been changed from 2 to 1.

27. Objective function #2 and the objChoice attribute are no longer documented in this guide. From this point forward, the documented version of optimizing implosion has only one objective function: objective #1, which is now called “the objective function”. For upward compatbility, objective #2 and the objChoice attribute still work as before; they are just no longer documented.

This applies to the following attributes and the corresponding API functions:

- problem.objChoice
- problem.capCost
- problem.invCost
- problem.periodsPerYear
- problem.obj2Wrev
- problem.obj2Winv
- problem.obj2Wserv
- problem.obj2Wsub
- problem.obj2RevValue
- problem.obj2InvValue
- problem.obj2ServValue

- problem.obj2SubValue
- part.unitCost
- demand.grossRev
- operation.obj2AuxCost
- subEntry.obj2SubPenalty

28. The names of the 6 attributes specific to objective #1 have been changed to reflect the fact that there is only one objective function. For upward compatibility, the API functions to set and retrieve the values of the old attributes still work, but they now set and retrieve the values of the corresponding new attributes. The data file format also accepts the old attribute names and sets the values of the corresponding new attributes when the old attribute names are used.

The correspondence is as follows

Object Type	Old Attribute	New Attribute
Part	obj1ScrapCost	scrapCost
Part	obj1StockCost	stockCost
Operation	obj1ExecCost	execCost
Demand	obj1CumShipReward	cumShipReward
Demand	obj1ShipReward	shipReward
Substitute	obj1SubCost	subCost

So, e.g., `witSetPartObj1ScrapCost` and `witSetPartScrapCost` both set the value of the scrapCost attribute.

29. The Post-Implosion Pegging algorithm now treats operations with multiple explodeable BOP entries in a different way. In the new treatment, if an operation has multiple explodeable BOP entries, the pegged execVol for the operation is allocated among its explodeable BOP entries in proportion to their pipShares. Explodeable BOP entries do not generate sideVol.

30. When WIT encounters an error condition, it can now optionally throw a C++ exception that can be caught and handled by the application program. To use this capability, the application program must be written in C++ and not in C.

There is one new message attribute for this capability:

- mesgThrowErrorExc

There are two new API functions for it:

- `witSetMesgThrowErrorExc`
- `witGetMesgThrowErrorExc`

And there is one new API data type for it:

- `WitErrorExc`

31. The WIT API can now be used in a multiply-threaded environment such as Java. Specifically, overlapping API function calls (i.e. those where one function call is initiated before the previous one is concluded) are now allowed, subject to certain restrictions.

There is one new global attribute for this capability:

- multiThread

There are two new API functions for it:

- `witSetMultiThread`
- `witGetMultiThread`

32. WIT API functions can now be called after WIT has issued a severe or fatal error message, subject to some restrictions.

33. A new kind of Post-Implosion Pegging has been added: “Group” Post-Implosion Pegging (GPIP). When an operation has multiple explodeable BOP entries, GPIP ignores the pipShare attribute and pegs all of the execVol of the operation through each of the explodeable BOP entries, so that each unit of execVol becomes pegged to a group of shipments instead of to an individual shipment.

The pre-existing form of PIP is now called “Individual” Post-Implsion Pegging (IPIP). On problems in which no operation has multiple explodeable BOP entries, GPIP and IPIP construct completely equivalent peggings.

There is one new global attribute for this capability:

- groupPipMode

There are two new API functions for it:

- `witSetGroupPipMode`
- `witGetGroupPipMode`

34. COIN-OR has replaced OSL as the LP solver for WIT.

The following items have been added:

- The global attribute solverLogFileName
- The API function `witSetSolverLogFileName`
- The API function `witGetSolverLogFileName`
- The control parameter solver_ofname

The following items have been removed from WIT:

- The global attribute oslMesgFileName
- The API function `witSetOslMesgFileName`
- The API function `witGetOslMesgFileName`
- The control parameter osl_ofname

Change History Release 7.0

1. The input data file format now accepts “7.0” in the release specification.
2. The Individual Post-Implosion Pegging capability (IPIP) has been removed. Group Post-Implosion Pegging (GPIP) is now called Post-Implosion Pegging (PIP).

   The following attributes were removed:

   - groupPipMode (global)
   - pipShare (for BOP entries)

   The following API functions were removed:

   - `witSetGroupPipMode`
   - `witGetGroupPipMode`
   - `witSetBopEntryPipShare`
   - `witSetBopEntryPipShareDbl`
   - `witGetBopEntryPipShare`
   - `witGetBopEntryPipShareDbl`
3. A new pegged quantity has been added to PIP: “coExecVol” (co-execution volume). For any BOP entry of an operation, any execution period, any demand, and any shipment period, the corresponding pegged coExecVol is the portion of the execVol of the operation in the execution period that’s pegged to the demand in the shipment period and whose pegging is specifically associated with the BOP entry.

   There are two new API functions for this capability:

   - `witGetDemandCoExecVolPip`
   - `witGetDemandCoExecVolPipDbl`
4. Two new global boolean attributes have been added: “nstnResidual” and “minimalExcess”. Setting nstnResidual to TRUE causes the residualVol attribute (on parts) to be computed differently. Setting minimalExcess to TRUE causes the excessVol attribute to be computed differently.

   There are four new API functions for these attributes:

   - `witSetNstnResidual`
   - `witGetNstnResidual`
   - `witSetMinimalExcess`
   - `witGetMinimalExcess`
5. A new float attribute for demands has been added: “searchInc”. When the heuristic attempts to meet a demand at less than the requested amount, it will only consider amounts that are multiples of searchInc (“search increment”) for the demand. Default Value: 1.0.

   There are four new API functions for this attribute:

   - `witSetDemandSearchInc`
   - `witSetDemandSearchIncDbl`
   - `witGetDemandSearchInc`

- `witGetDemandSearchIncDbl`

6. A new capability has been added to optimizing implosion: MIP mode, which causes optimizing implosion to be solved as a Mixed Integer Programming (MIP) problem.

There are four new attributes for this capability (all boolean):

- mipMode (global)
- intExecVols (operation)
- intSubVols (substitute)
- intShipVols (demand)

When mipMode is TRUE, optimizing implosion is solved as a MIP problem. When the intExecVols attribute for an operation is TRUE, optimizing implosion will constrain the execVol of the operation in all periods to take on integer values only. intSubVols and intShipVols are interpreted similarly.

MIP mode is an experimental aspect of WIT and should be used with caution: The resulting MIP problem may take vastly longer to solve than the corresponding LP problem without integrality constraints.

There are 8 new API functions for this capability:

- `witSetMipMode`
- `witGetMipMode`
- `witSetOperationIntExecVols`
- `witGetOperationIntExecVols`
- `witSetSubsBomEntryIntSubVols`
- `witGetSubsBomEntryIntSubVols`
- `witSetDemandIntShipVols`
- `witGetDemandIntShipVols`

7. An experimental new capability has been added to optimizing implosion: external optimizing implosion. Normally optimizing implosion involves a “solve” step in which WIT makes calls to solvers from COIN-OR to solve the LP/MIP formulation of the optimizing implosion problem. In external optimizing implosion, this “solve” step is performed by the application program: The application program extracts the LP/MIP problem from WIT, invokes its own solver to solve the LP/MIP problem and then loads the solution to the LP/MIP problem back into WIT. The external optimizing implosion capability consists of a set of API functions to perform the following tasks:

- Control the state of the external optimizing implosion process.
- Extract the LP/MIP problem.
- Load in the solution to the LP/MIP problem.
- Determine the column or row index of each variable or constraint in the LP/MIP problem, specified in terms of the objects of the implosion problem (parts, operations, etc.).

There is one new attribute for this capability:

- extOptActive (boolean, global, output only)

There are 25 new API functions for this capability:

- `witStartExtOpt`
- `witFinishExtOpt`
- `witShutDownExtOpt`
- `witGetExtOptActive`
- `witGetExtOptLpProb`
- `witGetExtOptLpProbDbl`
- `witGetExtOptIntVarIndices`
- `witSetExtOptSoln`
- `witSetExtOptSolnDbl`

- `witGetPartScrapVarIndex`
- `witGetPartStockVarIndex`
- `witGetDemandShipVarIndex`
- `witGetDemandCumShipVarIndex`
- `witGetOperationExecVarIndex`
- `witGetBomEntryNonSubVarIndex`
- `witGetSubsBomEntrySubVarIndex`
- `witGetPartStockSlbvVarIndex`
- `witGetDemandCumShipSlbvVarIndex`
- `witGetOperationExecSlbvVarIndex`
- `witGetPartResourceConIndex`
- `witGetDemandShipConIndex`
- `witGetBomEntrySubConIndex`
- `witGetPartStockSlbConIndex`
- `witGetDemandCumShipSlbConIndex`
- `witGetOperationExecSlbConIndex`

8. Six new API functions have been added for copying the input data attributes from one data object into another data object of the same type:

- `witCopyPartData`
- `witCopyDemandData`
- `witCopyOperationData`
- `witCopyBomEntryData`
- `witCopySubsBomEntryData`
- `witCopyBopEntryData`

9. Ordinarily, late shipments are allowed in an implosion solution. A new capability has been added that optionally disallows late shipments.

There is one new attribute for this capability:

- shipLateAllowed (boolean, by demand)

And there are two new API functions for it:

- `witSetDemandShipLateAllowed`
- `witGetDemandShipLateAllowed`

10. A new capability has been added that optionally disallows scrapping of selected parts. This applies only to optimizing implosion.

There is one new attribute for this capability:

- scrapAllowed (boolean, by part)

And there are two new API functions for it:

- `witSetPartScrapAllowed`
- `witGetPartScrapAllowed`

11. The API function `witOptPreprocess` now has the same effect as a call to `witPreprocess` and is no longer documented in this guide.

12. A new boolean input attribute for material parts has been added: asapPipOrder. When asapPipOrder is TRUE for a part, PIP will use the ASAP (As-Soon-As-Possible) pegging order on the part; when it is FALSE, PIP will use the NSTN (No-Sooner-Than-Necessary) pegging order on the part. (The default is FALSE.) The NSTN pegging order pegs the supplies, production, and side-effects of the part in later periods before proceeding to earlier periods, while the ASAP pegging order pegs in earlier periods before proceeding to later periods.

There are two new API functions for this capability:

- `witSetPartAsapPipOrder`
- `witGetPartAsapPipOrder`

13. A new form of implosion has been added to WIT: “stochastic implosion”. Stochastic implosion is a variant of optimizing implosion in which some of the input data is treated probabilistically, resulting in an implosion problem structured as a two-stage stochastic linear programming problem with recourse.

There are 8 new attributes for this capability:

- nScenarios
- stageByObject
- stochMode
- stochSolnMode
- objectStage (for parts)
- objectStage (for operations)
- currentScenario
- probability

And there are 19 new API functions for it:

- `witStochImplode`
- `witClearStochSoln`

- `witSetNscenarios`
- `witGetNscenarios`
- `witSetStageByObject`
- `witGetStageByObject`
- `witSetStochMode`
- `witGetStochMode`
- `witGetStochSolnMode`
- `witSetPartObjectStage`
- `witGetPartObjectStage`
- `witSetOperationObjectStage`
- `witGetOperationObjectStage`
- `witSetCurrentScenario`
- `witGetCurrentScenario`
- `witSetProbability`
- `witSetProbabilityDbl`
- `witGetProbability`
- `witGetProbabilityDbl`

14. A new API function, `witGetExpCycle`, has been added. This function searches for explodeable cycles in the complete BOM structure and retrieves one of them, if there are any. It does this without issuing an error message for the cycle.

15. The following attributes have now been made scenario-specific for stochastic implosion:

- scrapCost (part)
- stockCost (part)
- shipReward (demand)
- cumShipReward (demand)
- execCost (operation)
- subCost (substitute)

16. The function `witBuildPip` may now be called while WIT is in stochastic solution mode.

17. `witStochImplode` may now be invoked when there are BOM entries and substitutes connecting operations in stage 0 to parts in stage 1.

18. The names of existing parts, demands, and operations can now be changed. There are 3 new API functions for this capability:

- `witSetPartPartName`
- `witSetDemandDemandName`
- `witSetOperationOperationName`

19. A new approach to specifying which aspects of a stochastic implosion problem belong to each stage has been added: “stage-by-period” mode. In stage-by-period mode, the stage to which a data item belongs is determined by the period with which the data item is associated. For example the supplyVol of a part in period 2 might belong to stage 0, while the supplyVol of the part in period 3 belongs to stage 1.

There is one new attribute for this capability:

- periodStage

And there are two new API functions for it:

- `witSetPeriodStage`
- `witGetPeriodStage`

Also, the global boolean attribute stageByObject may now be given either boolean value:

- FALSE, for stage-by-period mode
- TRUE, for stage-by-object mode (which was the mode previously implemented)

20. New API functions for retrieving the values of the objValue and boundsValue attributes were added:

- `witGetObjValue`
- `witGetObjValueDbl`
- `witGetBoundsValue`
- `witGetBoundsValueDbl`

The following API functions were the pre-existing way of retrieving the value of these attributes:

- `witGetObjValues`
- `witGetObjValuesDbl`

These functions still exist, but are no longer documented in this Guide.

21. The default value of the message attribute mesgFiieAccessMode has been changed from “a” to “w”. Thus, by default, when WIT opens its message file, it opens it in overwrite mode.

22. The global multiThread attribute has been removed. WIT now behaves all the time as if the multiThread attribute were TRUE. The following API functions were removed:

- `witSetMultiThread`
- `witGetMultiThread`

23. A new capability has been added to optimizing implosion: “multiple objectives mode”. This capability allows more than one objective function to be specified and causes optimizing implosion to find an implosion solution that jointly maximizes all of the objective functions, treated as a strict hierarchy. There are 6 new attributes for this capability, all global:

- multiObjMode
- objectiveList
- objectiveListSpec
- currentObjective
- objectiveRank

- multiObjTol

And there are 12 new API functions for setting and retrieving the values of these new attributes.

24. A new capability has been added to heuristic implosion and allocation: “selection recovery”. This capability applies to each of the “selection” techniques of the heuristic: multiple routes, proportionate routing, build-ahead, stock reallocation, and multiple execution periods. When selection recovery is in effect, at the end of each increment to the heuristic allocation, all discarded selections (e.g., period-ahead periods) are recovered for use in the subsequent execution of the heuristic.

There is one new attribute for this capability, “selectionRecovery”.

25. A new capability has been added to heuristic implosion and allocation: “lead time bounds”. This capability allows an upper bound to be imposed on the time interval between the period in which a part is produced in order to enable a shipment and the period of the shipment. There are two new attributes for this capability:

- leadTimeUB: A vector of integers for each demand
- boundedLeadTimes: A boolean for each part

26. A new capability has been added to heuristic allocation: “modifiable heuristic allocation”. This capability allows the application program to make feasible modifications to the implosion solution during heuristic allocation. There is one new attribute for this capability:

- modHeurAlloc: A global boolean

27. A new capability has been added to optimizing implosion: optimization with CPLEX. This capability provides the option to have optimizing implosion solve its LP or MIP problem by invoking CPLEX instead of COIN. There are 4 new attributes for this capability:

- coinEmbedded: A global output boolean
- cplexEmbedded: A global output boolean
- coinSelected: A global input boolean
- cplexSelected: A global input boolean

28. The attributes intExecVols, intSubVols, and intShipVols are now allowed to be TRUE when mipMode is FALSE. When mipMode is FALSE, these attributes are ignored. (This allows an application program to easily switch between a MIP problem and its LP relaxation.)

29. A new capability has been added to optimization with CPLEX: CPLEX parameter specifications. This capability enables the application program to specify values for the parameters of CPLEX, which, in turn, allow the behavior of CPLEX to be customized. There are 7 new non-attributeAPI functions for this capability:

- `witAddIntCplexParSpec`
- `witAddDblCplexParSpec`
- `witAddDblCplexParSpecDbl`

- `witGetIntCplexParSpec`
- `witGetDblCplexParSpec`
- `witGetDblCplexParSpecDbl`
- `witClearCplexParSpecs`

There are also 3 new global input attributes for this capability:

- cplexParSpecName
- cplexParSpecIntVal
- cplexParSpecDblVal

30. Four new global output attributes have been added that provide information about the solution state of CPLEX:

- cplexStatusCode (type int): This is the CPLEX solution status code for the most recent call to a CPLEX solve routine, if any.
- cplexStatusText (type string): This is the CPLEX solution status text for the most recent call to a CPLEX solve routine, if any.
- cplexMipBound (type float): When the CPLEX MIP solver has been invoked, cplexMipBound is the tightest upper bound that it found on the optimal objective function value.
- cplexMipRelGap (type float): When the CPLEX MIP solver has been invoked, cplexMipRelGap is the relative gap between the objective function value and the tightest upper bound that it found on the optimal objective function value.

31. COIN-OR has been removed as a solver for WIT’s optimization problem. The following 3 attributes have been removed:

- coinEmbedded
- coinSelected
- cplexSelected

And the API functions to set and retrieve these attributes have been removed.

Change History Release 8.0

1. In the input data file format, the release specification is now optional and is not used for anything. Also, if a release specification is given, it is allowed to be any string.
2. The API function `witGetDemandExists` has been added. This function allows the application program to determine whether or not a specified demand exists in a WitRun.
3. The name of the global attribute “objectiveRank” was changed to “objectiveSeqNo”.
4. An extension the the post-implosion pegging capability has been added: “PIP to operations”. This capability considers each resource allocated in the solution and designates a set of operations and periods to which that resource is considered to have been allocated. It provides an answer to the question: "Which resources were used to enable a particular operation to be executed in a particular period?"

   There are 2 new attributes for this capability:

   - pipEnabled: An input boolean attribute of operations
   - pipRank: An input int attribute of operations

   And there are 18 new API functions for it:

   - witSetOperationPipEnabled
   - witGetOperationPipEnabled
   - witSetOperationPipRank
   - witGetOperationPipRank
   - witGetOperationSupplyVolPip
   - witGetOperationExecVolPip
   - witGetOperationSubVolPip
   - witGetOperationProdVolPip
   - witGetOperationConsVolPip
   - witGetOperationCoExecVolPip
   - witGetOperationSideVolPip
   - witGetOperationSupplyVolPipDbl
   - witGetOperationExecVolPipDbl
   - witGetOperationSubVolPipDbl
   - witGetOperationProdVolPipDbl
   - witGetOperationConsVolPipDbl
   - witGetOperationCoExecVolPipDbl
   - witGetOperationSideVolPipDbl

CHAPTER 1 Introducing WIT

"His foe was folly and his weapon wit."

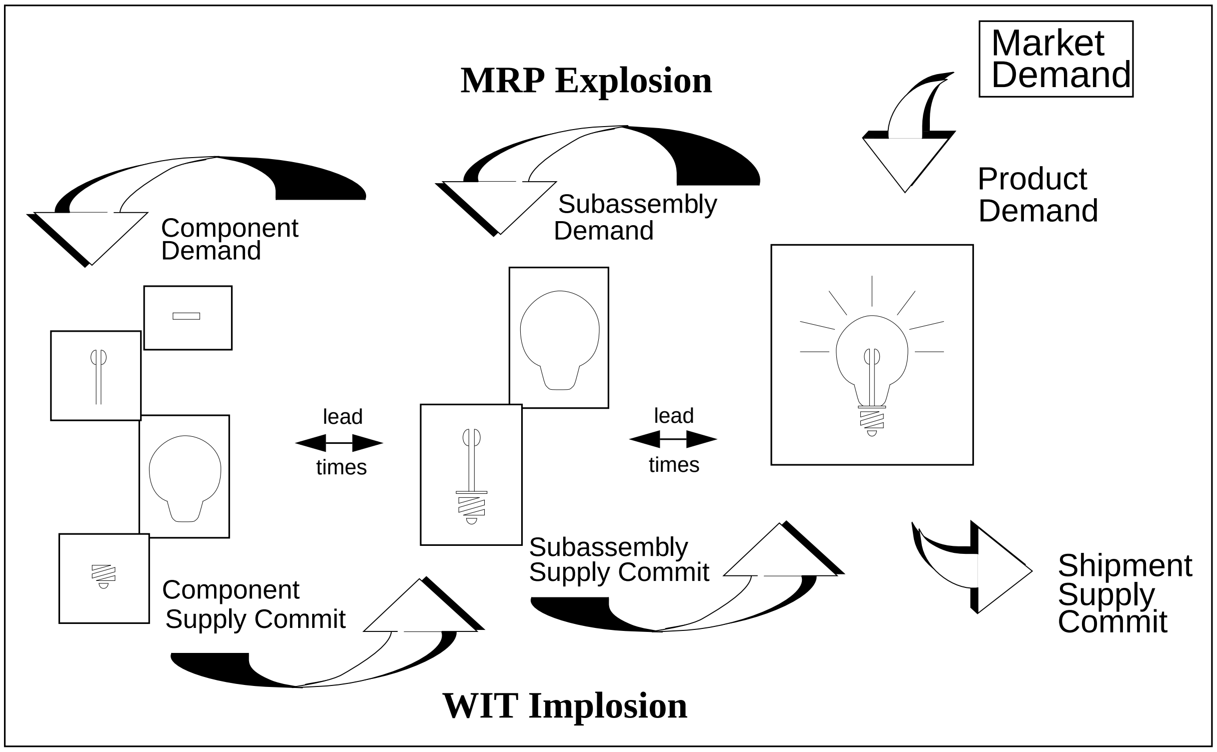
Anthony Hope
Inscription on the tablet to W.S. Gilbert
Victoria Embankment, London
1915

What is WIT?

Constrained Materials Management and Production Planning Tool (WIT) is a software tool for constrained materials management and production planning. The main data for this software tool is a list of demands for products, a list of supplies for components, and a multilevel bill-of-manufacturing (BOM) describing how to build products from components. Ordinarily, it takes many components to build a single product, and so the list of supplies is much larger than the list of demands. A traditional use of a BOM to perform production planning is Material Requirements Planning (MRP), in which the list of demands for products is "exploded" (via the BOM) into a much larger list of requirements for components. The main capability provided by WIT is, in some sense, the reverse of an MRP explosion: the WIT "implosion". The list of supplies of components is "imploded" via the BOM into a relatively small list of feasible shipments of demanded products. The assumption is that not all demands can be met, and so the idea is to make judicious trade-offs between the different demands given limited supplies to best satisfy the manufacturing objectives. (**Historical note:** This software was originally called "Workstation Implosion Tool", with an acronym WIT. Since then it has undergone several name changes. Its current name is Constrained Materials Management and Production Planning Tool, but it is informally called WIT. WIT is no longer an acronym.)

The following diagram illustrates the relationship between MRP explosion and WIT implosion.

FIGURE 1 **The Relationship Between MRP Explosion and WIT Implosion**



As an alternative to solving the implosion problem, WIT can optionally perform a limited, but very fast form of Material Requirements Planning, called WIT-MRP. The WIT-MRP takes a set of bills of material and a set of demands and explodes them to get a requirements schedule for supplies.

WIT has been designed to be used in either of two modes: stand-alone mode and API mode. In stand-alone mode, the user runs a program, the WIT stand-alone executable, which communicates by files and performs WIT’s main actions: implosion and WIT-MRP. The WIT Application Program Interface (API) is a collection of C functions that perform each of WIT's actions, including implosion, WIT-MRP, data communication by files and data communication in memory. In API mode, a programmer writes an application program in C that calls WIT's functions directly. The end-user then runs the application program. In most cases, WIT is used in API mode, because the API is more flexible and allows tighter integration with the environment in which WIT is to be used. However, stand-alone mode can frequently be useful as means of learning WIT and as a “rapid prototypes facility” for WIT, because it allows users to try out their implosion ideas quickly without programming.

See also [2] for a fairly detailed and comprehensive introduction to WIT.

Terminology

The following list defines manufacturing terms as they are used in this guide. This information is presented up front to provide a common background when discussing WIT.

System	The manufacturing enterprise being modeled by WIT.
Period	The time bucket used by WIT for its production plan. Normally, one week, but it could be shorter or longer.
Part	WIT's concept of a part is very general. A part is either a product or anything that is consumed in order to build a product. This includes both materials and capacities.
Operation	An operation is the means by which parts are built. Specifically, a part is built by "executing" an operation. When an operation is executed, some parts are consumed, while other parts are produced.
Part Category	Parts are classified into two categories: • Material • Capacity
Material	A material part is either a raw material or a product. It usually represents an actual physical object, in contrast to a capacity, which usually represents available time (either human time or machine time). The defining property of a material is that any quantity of a material part that is not used in one period remains available in the next period. In other words, material parts have stock (i.e., inventory).
Capacity	A capacity represents some limitation on the quantity of one or more operations that can be executed during one period. The defining property of a capacity is that any quantity of a capacity that is not used in one period is lost and is not available in the next period. In other words, capacities don't have stock. In the case of capacity, external supply in a given period is interpreted to mean the available quantity of the

	capacity in that period. For a further discussion, see “Capacity” on page 9.
BOM (Bill-of-Manufacturing)	Each operation has a bill-of-manufacturing that specifies the how parts (both material and capacity) are consumed when the operation is executed. Specifically, it is a list of “BOM entries” indicating which parts are consumed, “offsets” indicating when the parts are consumed, usage rates indicating how much of the material or capacity is consumed, and so on. In MRP terms, this is roughly equivalent to a combined bill-of-material and bill-of-capacity.
BOM entry	A BOM entry is the association between a particular operation and one particular part in its BOM. Each BOM entry represents the consumption of some volume of a part in order to execute some operation.
Substitute BOM entry	Each BOM entry may optionally have one or more substitute BOM entries associated with it. Each substitute BOM entry represents an alternative part to be consumed in place of the consumed part indicated by the original BOM entry.
Substitute	Same as substitute BOM entry.
Component	Any part that appears as the consumed part of a BOM entry or substitute BOM entry. A component may be either a material or a capacity.
BOP (Bill-of-Products)	Each operation has a bill-of-products that specifies the how parts are produced when the operation is executed. Specifically, it is a list of “BOP entries” indicating which parts are produced, “offsets” indicating when the parts are produced, production rates indicating how much of the part is produced, and so on.
BOP entry	A BOP entry is the association between a particular operation and one particular part in its BOP. Each BOP entry represents the production of some volume of a part as a result of executing

	some operation. Usually, the produced part will be a material part, but WIT also allows the produced part to be a capacity, in order to accommodate unusual modeling situations.
Product	Any part that appears as the produced part of a BOP entry.
Demand	Each material part may optionally have one or more demands (also known as “demand streams”) associated with it. A demand stream represents an external customer who places demands for the part. (“External customer” means external to the system being modeled by WIT. So it is possible for an external customer to be internal to the corporation.) More than one demand stream is permitted for the same part, to represent the fact that some demand for a part may be more urgent than other demand for the same part. It is also possible for a part to have no demand streams, if it is only needed for its role as a resource to be consumed by some operation.
Demand Stream	Same as Demand.
Complete BOM structure	The complete interrelationship between all parts and operations that arises when one considers all of the BOM entries, substitutes, and BOP entries in the problem.
Execution Schedule	This is a specification of how much of each operation is to be executed during each period. If there are substitutes, the Execution Schedule also specifies how much of the operation’s execution in each period is due to each substitute in its BOM.
Shipment Schedule	This is a specification of how much of each part is to be shipped to each demand stream during each period.
Implosion Solution	The implosion solution is simply the Execution Schedule and Shipment Schedule taken together. This is the main output of WIT. The Shipment Schedule shows to what extent the demands can be met, and the Execution Schedule shows how the Shipment Schedule can be achieved.

Requirements Schedule	This is the main output of WIT-MRP. It is a specification of how much additional supply of each part is required in each period in order to meet all demands on time.

Single Method Assembly Problem

A WIT problem in which:

- Each part appears as the produced part of at most one BOP entry.
- Each operation has exactly one BOP entry.
- There are no capacity products.

Thus in a single method assembly problem, there is a one-to-one correspondence between operations and products, where each product is a material that can be built in exactly one way (except by using substitutes), and this one way produces no other product. This is the situation modeled by MRP systems and by versions of WIT prior to release 4.0. Many WIT problems are single method assembly problems.

Single method assembly problems are also called SMA problems. Problems that aren't single method assembly problems are called non-SMA problems.

Example WIT Problem Diagrams

It is sometimes useful to illustrate a WIT problem with a diagram. The symbols used in these diagrams are shown in Figure 2.

FIGURE 2 Symbols Used in WIT Problem Diagrams

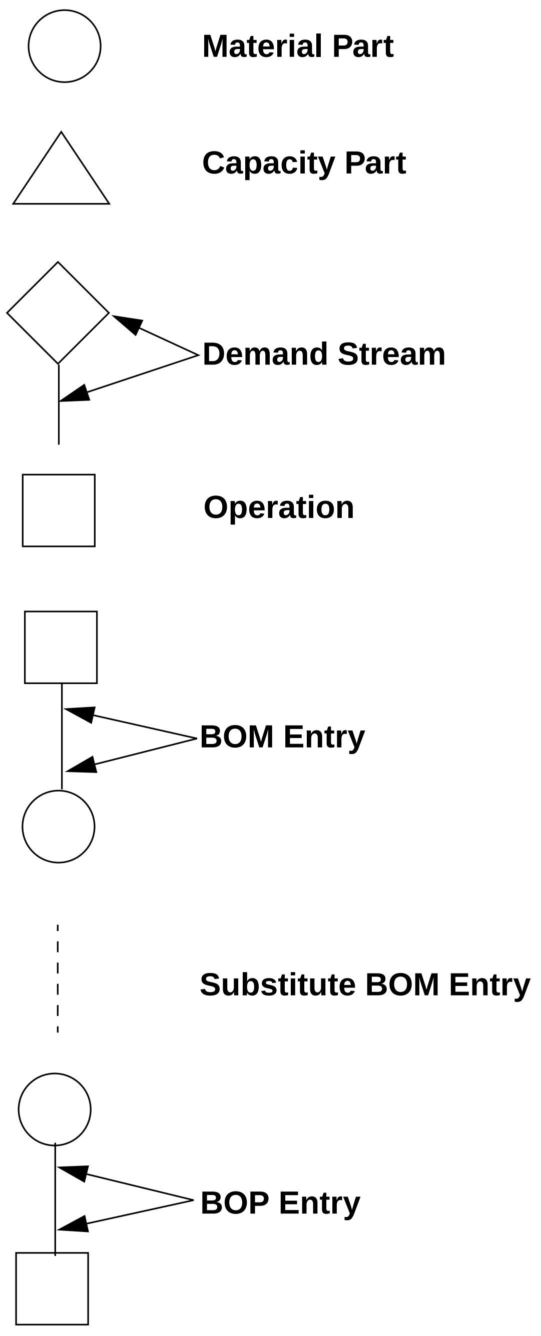
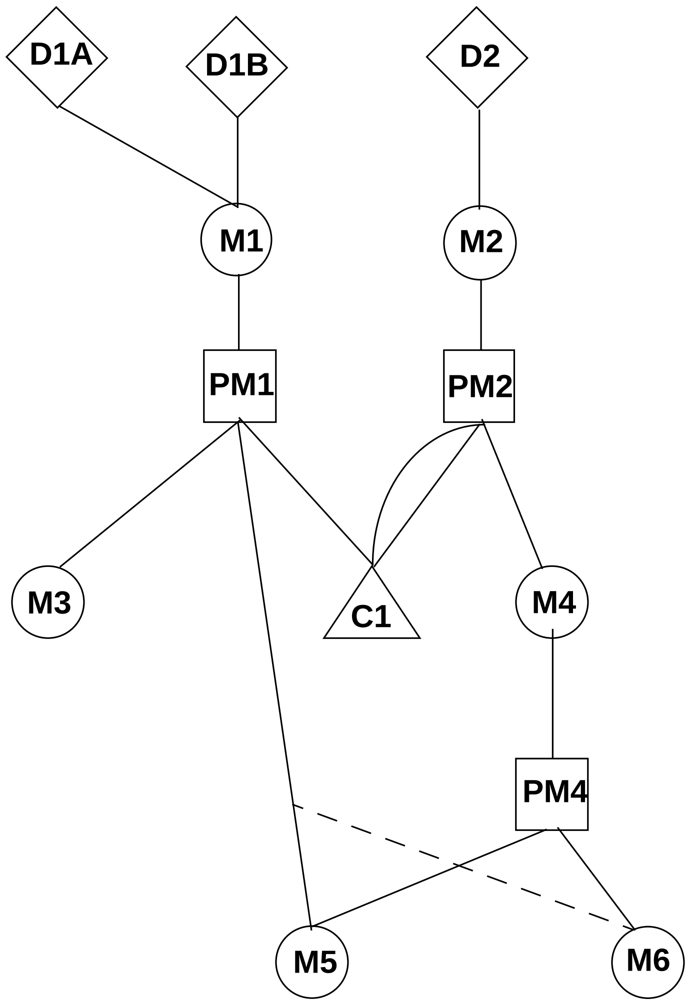


Figure 3 illustrates an example of an SMA problem.

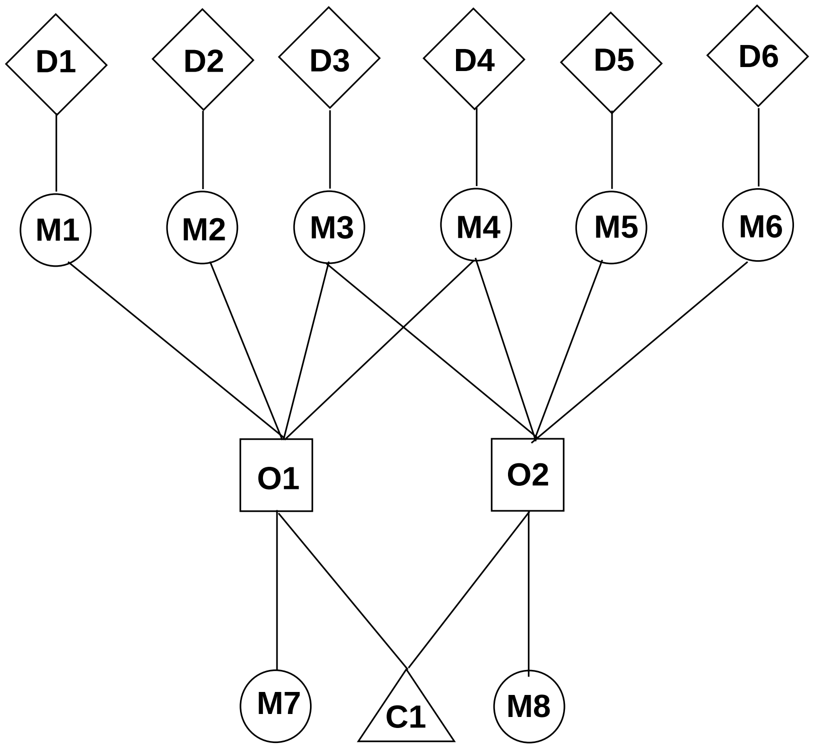
FIGURE 3 A Single Method Assembly Problem



In the problem diagrammed in Figure 3, there are 7 parts. M1, M2, M3, M4, M5, and M6 are material parts and C1 is a capacity. There are 3 operations, PM1, PM2, and PM4. As indicated by the BOP entries, PM1 produces M1, PM2 produces M2, and PM4 produces M4. There are 3 products, M1, M2, and M4. Since this is a single method assembly problem, there is a one-to-one correspondence between the products and the operations that produce them. For example, M2 is produced only by PM2 and PM2 produces only M2, so one can say that PM2 is the operation for producing M2. As indicated by the BOM entries, PM1 consumes M3, M5, and C1, PM2 consumes C1 and M4, and PM4 consumes M5 and M6. The substitute BOM entry indicates that M6 can be consumed in place of M5 when executing PM1. There are 2 BOM entries associating PM2 with C1, indicating (perhaps) consumption of C1 at two different times during the execution of PM2. There are 3 demand streams, D1A, D1B, and D2. D1A and D1B are demands for M1, and D2 is a demand for M2.

Figure 4 illustrates an example of a non-SMA problem. Each operation produces 4 parts and there are 2 parts (M3 and M4) each of which can be produced by 2 different operations.

FIGURE 4 A Non-SMA Problem



Capacity

As stated on page 3, a capacity part represents some limitation on the quantity of one or more operations that can be executed during one period.

Consider the following example: The execution of some operation is limited by the fact that it requires processing time on machine class X. Assume there are 3 machines of class X, and they are available 70% of the time and a period presents a 14-shift work week with 8 hour shifts. To represent this limitation, a capacity corresponding to “machine hours of machine class X” is defined. The external supply of this capacity is 235.2 (= 3 * 0.7 * 14 * 8) machine hours per period. Any operation that requires processing on this machine should have an entry for “machine hours of machine class X” in its BOM and with a usage rate equal to the number of hours of processing required on machine class X.

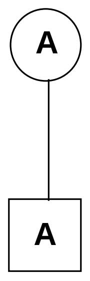
NOTE: The use of the word “part” to describe capacity may at first seem rather surprising. It’s clear that a material part has some rather different properties from, say, an hour of machine time. Internally, WIT carefully models all of the essential differences between material parts and capacities. However, we have found that there are enough similarities between material parts and capacities that we could make communication between WIT and its users simpler and more compact, by applying the same terminology to both. For example, you can specify external supply for any kind of part in WIT. If the part is a material part, external supply has its ordinary meaning. If it is a capacity, external supply is interpreted to mean the available quantity of the capacity in a given period.

Given that one word was to be used to designate both material parts and capacity, we chose the word “part”, because one can think of a capacity as a “part” of the product to which it contributes.

Part-With-Operation

The term “part-with-operation” denotes a cluster of 3 data objects: a material part, an operation with the same name as the part and a BOP entry connecting the part with the operation. The following is a diagram of a part-with-operation.

FIGURE 5 A Part-With-Operation



A part-with-operation is a natural model for when a material can only be produced by one operation and that operation produces only that part. This occurs commonly not only (by definition) in single method assembly problems, but in other problems as well. To facilitate this, WIT provides an explicit ability to add part-with-operations, either with a direct API function or with the input data file.

WIT Input Overview

In order to better understand WIT and its methods, the following list of input data is presented. Complete data descriptions are found in Chapter 2. Data is specified to WIT using either a data file or through the API. Most input data can be defaulted if it is not relevant to the user's problem.

Global Data

- Specification of the planning horizon
- Selection of an Objective choice (more about Objectives at a later time)

Part Data

- The part name
- The part category: Material or Capacity
- The standard unit cost of the part
- External supply volumes for each period
- Holding costs
- Scrapping costs

Demand Data

- Demand stream name
- The gross revenue per unit shipped to the demand stream
- Priority for each period
- Demand volume for each period
- Shipment reward
- Cumulative shipment reward

Operation Data

- The operation name
- Yield rates for each period
- Execution cost

BOM Entry Data

- The consuming operation
- The consumed part
- Offset
- Usage rate
- Fallout rate
- Effectivity intervals (earliest/latest)

- Mandatory engineering change option

Substitute BOM Entry Data

- The associated BOM entry
- The component consumed
- Offset
- Usage rate
- Fallout rate
- Effectivity intervals (earliest/latest)
- The penalty on the use of each substitute

BOP Entry Data

- The producing operation
- The produced part
- Offset
- Production rate
- Effectivity intervals (earliest/latest)

Bounds

Additionally, WIT allows the user optionally to specify bounds on the following:

- Stock levels
- Cumulative shipment volume
- Execution volume

Implosion Methods Used

WIT solves the implosion problem using one of three different methods, at the user's discretion:

- Heuristic implosion
- Optimizing implosion
- Stochastic Implosion

The distinction between these methods stems from the fact that there are many possible implosion solutions, but some solutions are better suited to the user's needs than others.

- Heuristic implosion finds a feasible implosion solution, guided by priorities set by the user.

- Optimizing implosion finds the “best possible” implosion solution, according to an objective function that defines the “goodness” of any possible solution.
- Stochastic implosion is similar to optimizing implosion: It also finds the “best possible” implosion solution, according to an objective function. However, it does this in the context of implosion problem in which some of the input data is initially not known with certainty and must be modeled probabilistically. In contrast, all of the input data for heuristic implosion and optimizing implosion is deterministic.

Some properties common to all three methods are:

1. The implosion solution will be feasible; that is, there is sufficient supply of material and capacity to meet the Execution Schedule and the Shipment Schedule.
2. Implosion will never plan to ship early; that is, for any demand stream `d` and any period `t`, the cumulative planned shipment toward demand `d` through period `t` will never exceed the cumulative demand `d` through period `t`.

Heuristic Implosion

Heuristic implosion (sometimes called “the heuristic”) computes an execution schedule and shipment schedule based on the priorities associated with each demand stream. Some properties of heuristic implosion are:

1. The heuristic uses lot sizing.
2. When the heuristic performs its internal explosion and netting, it nets from the top of the BOM down (e.g. won't build if it has sufficient stock.)
3. Unless one of the build-ahead options is being used, the heuristic will try to produce “just-in-time” to meet demand. (See also “Build-Ahead” on page 82.) If sufficient resources are not available to produce just-in-time, it will produce late. Specifically, if demand `d` requires `q` units of product in period `t`, the heuristic first determines the quantity `p` that can be completed in period `t` and allocates resources to meet this production. If the heuristic is not able to meet all of the demand on time, it will try to meet the remaining demand in the later periods `t+1, t+2, ...`., until either the demand is met, the end of the problem horizon is reached, or the shipLateUB for demand `d` is reached. (The shipLateUB is a demand attribute defined in chapter 2.)
4. Much of the behavior of the heuristic can be understood by considering the trade-offs it would make in the following situation: Suppose the heuristic is deciding between meeting the portion of the demand volume for demand stream d_1 incurred in demand period dt_1 by shipping in shipment period st_1, versus meeting the portion of the demand volume for demand stream d_2 incurred in demand period dt_2 by shipping in shipment period st_2. (Note that we must have $st_1 \geq dt_1$ and $st_2 \geq dt_2$.) Then:

5. If the demand periods are not the same ($dt_1 \neq dt_2$), the heuristic will give preference to meeting the demand volume with the earlier demand period.
6. If the demand periods are the same, but the shipment periods are different ($dt_1 = dt_2$, $st_1 \neq st_2$), the heuristic will give preference to the demand volume that is to be met in the earlier shipment period.
7. If the demand periods are the same and the shipment periods are the same ($dt_1 = dt_2$, $st_1 = st_2$), the heuristic will give preference to the demand with the higher priority in the demand period.
8. If the demand periods are the same, the shipment periods are the same, and the priorities are the same, the heuristic will give preference to the demand that was entered earlier into WIT. (This is the final tie-breaker.).
9. Ordinarily, the heuristic uses available supply of substitutes, but does not build parts to use as substitutes. This behavior can be overridden: See “Multiple Routes” on page 78. If a BOM entry has more than one possible substitute, the heuristic considers the possible substitutes in the order in which they were input to WIT.
10. Ordinarily, if a part appears as the produced part of more than one BOP entry, indicating that there are multiple ways to produce the same part, the heuristic will not exploit this. Instead, it will choose one BOP entry and, whenever more quantity of the part is needed, it will execute the producing operation for that BOP entry only. However, this behavior can be overridden: See “Multiple Routes” on page 78.
11. If an operation has more than one BOP entry, the heuristic will tend not to take advantage of this. Instead, it will execute the operation in order to produce whichever part it is considering at the moment, regardless of whether or not the other produced parts are needed.
12. The heuristic ignores most kinds of bounds, with certain exceptions:
    - Hard upper bounds on execution volumes. The heuristic always respects these.
    - Soft lower bounds on stock volumes. By default, the heuristic ignores these, but it will respect them, if the “respectStockSLBs” attribute is set to TRUE. See “respectStockSLBs” on page 113. Also, when there are resource conflicts between satisfying stock soft lower bounds and meeting demands, the heuristic will resolve them in favor of the demands.
    - All other kinds of bounds are ignored by the heuristic.
13. Also available is the Equitable Allocation Heuristic Implosion. For more information see “Equitable Allocation Heuristic Implosion” on page 69.

Optimizing Implosion

Optimizing implosion finds the best possible implosion solution according to some well-defined criterion. The criterion for determining the “goodness” of a proposed implosion solution is specified as an objective function. This is a mathematical function that takes an implosion solution as its input and converts it into a single number that defines the goodness of the implosion solution.

Optimizing implosion normally proceeds as follows: The input data is translated into a Linear Programming (LP) formulation of the implosion problem. This LP is then solved by invoking general-purpose optimization software that has been incorporated into WIT. The solution to the LP is then translated into an implosion solution. A general description of WIT's LP formulation can be found in [1]. For a precise specification of WIT's LP formulation, see [3].

WIT solves its LP formulation by invoking IBM ILOG CPLEX. CPLEX is IBM's software for solving optimization problems including linear and mixed integer programming. For information on CPLEX, see:

```
http://www-01.ibm.com/software/integration/optimization/
      cplex-optimizer/
```

Some aspects of optimizing implosion that distinguish it from heuristic implosion are:

1. An optimal implosion solution is produced.
2. WIT usually takes much more CPU time and more memory to compute the implosion solution using optimizing implosion than when using heuristic implosion.
3. Optimizing implosion does not do lot sizing.
4. Optimizing implosion does not do Equitable Allocation.
5. Optimizing implosion respects all upper and lower bounds.
6. Optimizing implosion is not restricted to building parts "just-in-time". It will build parts earlier than needed, if this is appropriate, e.g., due to capacity limitations. This applies both when more volume of the part is needed for demand streams associated with the part as well as when more volume the part is needed in order for it to be consumed by an operation.
7. Optimizing implosion will build parts to be used as substitutes, as needed. The order in which substitutes are input to WIT is not important to optimizing implosion.
8. Optimizing implosion is not allowed, if the problem contains no parts. (This restriction is actually applied during preprocessing for optimizing implosion.)
9. At the user's request, optimizing implosion will compute "shadow prices" for the parts. See "shadowPrice" on page 124 and "compPrices" on page 99.

The Primary Objective Function

WIT's "objective function" is a mathematical function that takes an implosion solution as its input and converts it into a single number that is taken to be the quality of the implosion solution. This section describes an essential version of the objective function, called the primary objective, primaryValue, which assumes that the implosion problem includes no soft lower bounds. After this, the extension to the objective function for soft lower bounds will be described.

The basic idea of the primary objective is to allow the user to set the cost and reward coefficients of each variable in the linear programming formulation. The data for the primary objective is as follows:

- For each material part, `j`, and each period, `t`:
  $\texttt{stockCost}_{j,t}$
  `float`

  This is the cost incurred for each unit of part `j` held in inventory at the end of period `t`.
- For each part, `j`, and each period, `t`:
  $\texttt{scrapCost}_{j,t}$
  `float`

  This is the cost incurred for each unit of part `j` scrapped in period `t`.
- For each demand stream, `i`, and each period, `t`:
  $\texttt{shipReward}_{i,t}$
  `float`

  This is the reward received for each unit shipped to demand stream `i` in period `t`.
- For each demand stream, `i`, and each period, `t`:
  $\texttt{cumShipReward}_{i,t}$
  `float`

  This is the reward received for each unit of cumulative shipment to demand stream `i` by the end of period `t`. (i.e., it is the reward received per unit for the total volume shipped to demand stream `i` in periods 0, 1,..., t.)
- For each operation, `h`, and each period, `t`:
  $\texttt{execCost}_{h,t}$
  `float`

  This is the cost incurred for each unit of operation `h` executed in period `t`.
- For each substitute BOM entry, `s`, and each period, `t`:
  $\texttt{subCost}_{s,t}$
  `float`
  Let operation `h` be the consuming operation for substitute BOM entry `s`. Then $\text{subCost}_{s,t}$ is the cost incurred for each unit of operation `h` that is executed in period `t` using substitute BOM entry `s`.

The primary objective is defined as follows:

`primaryValue =`

$$- \sum_{j,t} \texttt{stockCost}_{j,t} * \texttt{stock volume of part j in period t}$$

(The sum is taken over all material parts, j, and all periods, t.)

$- \sum_{j,t}$ scrapCost$_{j,t}$ * scrap volume of part j in period t

(The sum is taken over all parts, j, and all periods, t.)

$+ \sum_{i,t}$ shipReward$_{i,t}$ * shipment volume to demand stream i in period t

(The sum is taken over all demand streams, i, and all periods, t.)

$+ \sum_{i,t}$ cumShipReward$_{i,t}$ * cumulative shipment volume to demand stream i in period t

(The sum is taken over all demand streams, i, and all periods, t.)

$- \sum_{h,t}$ execCost$_{h,t}$ * execution volume of operation h in period t

(The sum is taken over all operations, h, and all periods, t.)

$- \sum_{s,t}$ subCost$_{s,t}$ * volume executed using substitute BOM entry s in period t

(The sum is taken over all substitutes s, and all periods, t.)

In the absence of soft lower bounds, the objective function is just:

objValue = primaryValue

Recommendations for the Primary Objective:

- We recommend that all costs in the primary objective be specified as ≥ 0.0. This applies to the following:
  - stockCost
  - scrapCost
  - execCost
  - subCost

  WIT displays a warning message the first time one of these costs is given as negative. You may specify these costs as negative, but if you do, you run the risk of defining an implosion problem that has no optimal solution, because the objective function can be made arbitrarily large, i.e., it goes to positive infinity. For example, suppose execCost < 0.0 for an operation, and the operation has an empty bill of manufacturing, and execEmptyBOM is TRUE, and scrapCost = 0.0. Then WIT can make the execution volume of this operation larger and larger, with no limit, while always making the objective function larger. When this condition occurs, WIT generates an error message indicating that the objective goes to infinity and it does not produce an implosion solution. If you get this error message, we can suggest two ways to proceed. One approach is to remove all negative costs. The objective cannot go to infinity if there are no negative costs. Another approach is to put finite hard upper bounds on execution; See “execBounds” on page 131. If all operations have finite hard upper bounds on execution, then the objective cannot go to infinity.

- For most implosion problems, we recommend specifying an objective function that provides an incentive to ship to each demand on time, if possible. Two ways to do this are:
  - Specify cumShipReward$_{i,t} > 0$ for all demands and periods, or
  - Specify shipReward$_{i,t}$ > shipReward$_{i,t+1} > 0$ for all demands and periods.

The Objective Function with Soft Lower Bounds

If the input data includes soft lower bounds (that are larger than the corresponding hard lower bounds), the objective function is extended by including a bounds objective, boundsValue, which imposes a penalty on violations of soft lower bounds. boundsValue is defined to be the sum of the violations of all soft lower bounds. The input data associated with this objective is as follows:

wbounds:	The weight on the bounds objective. It is specified as real number ≥ 0.0. High values give more importance to the bounds objective; low values give less importance to it. A value of 0.0 will cause WIT to ignore the bounds objective and therefore ignore all soft lower bounds. See also “wbounds Calculation and Recommendation” on page 20.

With soft lower bounds, the objective function is defined as follows:

objValue = primaryValue - wbounds $*$ boundsValue

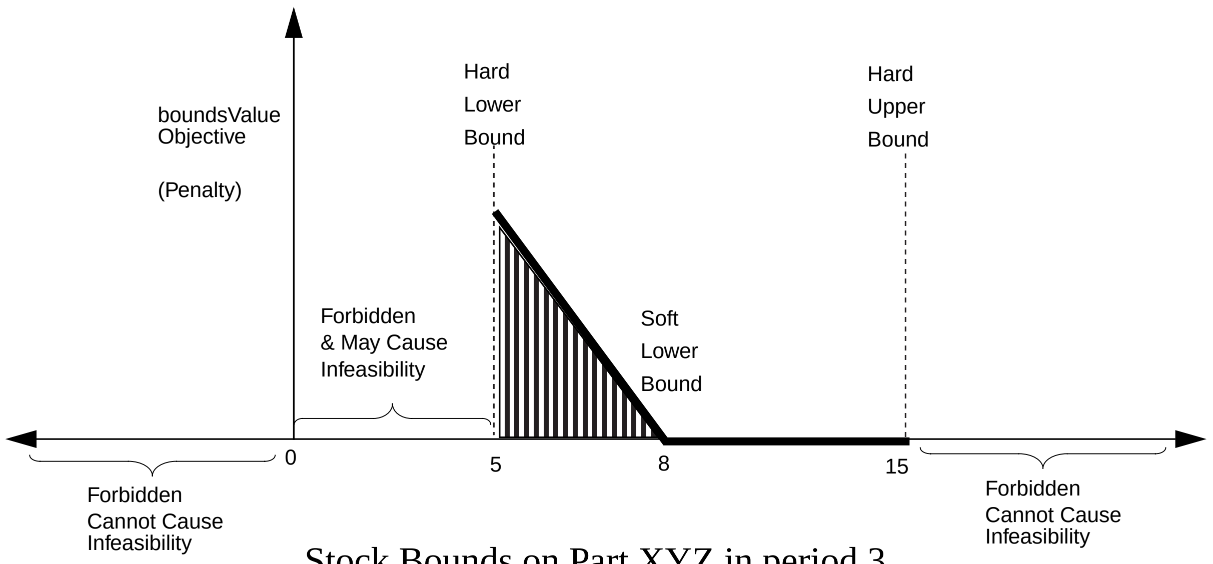
Bound Set Recommendations

Violations of the hard lower bounds are forbidden, so the resulting implosion problem may be infeasible, meaning no solution exists that is consistent with the hard lower bounds. If WIT determines a problem is infeasible, it issues an error message and doesn’t return a solution. This puts the user in the unfortunate position of knowing that the hard lower bounds cannot be satisfied, but not knowing which ones (or combinations) to change. We recommend that only experienced WIT users use hard lower bounds.

We strongly encourage most users to:

1. Set the hard lower bounds to 0.0
2. Specify lower bound requirements only as soft lower bounds. This approach is guaranteed to have a feasible solution.

FIGURE 6 Bound Set Illustration



boundsValue Objective
(Penalty)
Hard Lower Bound
Hard Upper Bound
Forbidden & May Cause Infeasibility
Soft Lower Bound
0
5
8
15
Forbidden Cannot Cause Infeasibility
Forbidden Cannot Cause Infeasibility

Stock Bounds on Part XYZ in period 3

Figure 6 illustrates an example of how bounds work. In this case, the bounds are on the stock levels of a particular part (XYZ) in a particular period (3). Figure 6 shows that the hard lower bound on the stock level of part XYZ in period 3 is 5. This means that when WIT is trying to find an optimal implosion solution, it will only consider implosion solutions that result in a stock level of 5 or more for part XZY in period 3. If it is impossible to construct any implosion solution whose stock level for part XZY in period 3 is 5 or more, then the implosion problem is considered to be infeasible and WIT gives a message to that effect.

The soft lower bound on the stock level of part XYZ in period 3 is 8. When WIT is trying to find an optimal implosion solution, it will certainly consider implosion solutions that result in a stock level 8 or more for part XZY in period 3, but it will also consider implosion solutions that result in a stock level of less than 8 for part XZY in period 3. However, when the stock level for part XZY in period 3 is less than 8, a penalty is applied to the objective function, by increasing boundsValue. This penalty increases linearly for stock levels that are in greater violation of the soft lower bound, i.e., further below 8. In contrast, when the stock level for part XZY in period 3 is 8 or more, no penalty is applied to the objective function. Thus, all other things being equal, WIT will prefer implosion solutions that result in a stock level of 8 or more for part XZY in period 3 to those that don't.

Finally, the hard upper bound on the stock level of part XYZ in period 3 is 15. This means that when WIT is trying to find an optimal implosion solution, it will only consider implosion solutions that result in a stock level of 15 or less for part XZY in period 3. Fortunately, optimizing implosion can always find a solution that satisfies the hard upper bounds.

We recommend that only experienced WIT users use positive hard lower bounds. We encourage most users to set the hard lower bounds to zero, because it is guaranteed to produce a feasible solution.

wbounds Calculation and Recommendation

The recommended value for wbounds is 10,000. The purpose of using a very high weight on the wbounds objective is to cause optimizing implosion to achieve the following result:

(1) Minimize the bounds objective, and

(2) Maximize the primary objective subject to achieving (1).

Clearly, if wbounds is set too small, (1) will not be achieved. However, if wbounds is set too large, then the rest of the objective will be numerical “noise” compared to the bounds objective and (2) will not be achieved. Fortunately, in tests we found that both (1) and (2) could be achieved for a wide range of values of wbounds. For a typical problem, we found that we could set wbounds anywhere from 10,000 to 1,000,000,000 and still achieve (1) and (2).

To calibrate wbounds for your type of implosion problem, you can use the following procedure:

1. Select a “typical” data set.
2. Set wbounds to 10,000.
3. Run WIT.
4. Multiply wbounds by 10.
5. Repeat 3 and 4 until the resulting boundsValue does not decrease.
6. Use this value of wbounds on all future runs of WIT.

Or if it is not very important to achieve (1), you could just set wbounds = 10,000.

Stochastic Implosion

Introduction

Please note: Stochastic implosion is an advanced feature of WIT. The following description assumes familiarity with (and ideally, experience with) optimizing implosion.

Stochastic implosion solves a variant of the optimizing implosion problem in which some of the input data is initially not known with certainty and must be modeled probabilistically. More precisely, the stochastic implosion problem is an implosion-specific version of the standard “stochastic linear programming with two stages and multiple scenarios” formulation. Detailed information on stochastic programming can be found in [1]; however, the explanation of stochastic implosion given here is intended to be self-contained.

Generally speaking, an implosion problem may be thought of as a set of planning decisions to be made: Given the input data, one must decide shipment volumes, execution volumes and substitution volumes. A deterministic implosion problem (i.e., heuristic or optimizing implosion) models a situation in which the decisions are to be made simultaneously and with complete knowledge of the values of all the input data attributes. A stochastic implosion problem models a situation in which the decisions are to be made in two distinct stages:

- Some of the decisions must be made during stage 0, in which much of the input data is known deterministically, but the values of some of the input data attributes are specified probabilistically, as random variables.
- The remaining decisions are to be made during stage 1, at which point the values of all probabilistic data are known with certainty.

One example of a stochastic implosion problem would be one in which all data is known deterministically except for the demand volumes, which are treated as random. The execution volumes of the lower level operations would have to be decided before the demand volumes are known with certainty, while the shipment volumes as well as the execution volumes of the upper level operations would not need to be decided until complete deterministic knowledge of the demand volumes has become available.

The probabilistic aspects of a stochastic implosion problem are specified via a finite set of stochastic “scenarios”. A scenario is a complete specification of the values of all of the probabilistic data in the problem. The set of all scenarios corresponds to the set of all possible outcomes of the random variables in the problem data. Each scenario has a probability associated with it and the sum of all scenario probabilities must be equal to 1.

More precisely, each attribute whose value is not known deterministically in stage 0 has a potentially distinct value associated with each scenario. These attributes are called “scenario-specific” attributes. For each scenario, s, there exists (conceptually) an associated “scenario s implosion problem”, defined by all the stage 0 data together with all the stage 1 data for scenario s. The scenario s implosion problem is equivalent to the ordinary optimizing implosion problem that would be specified by the stage 0 data and the stage 1 data for scenario s.

A stochastic implosion problem is a model of the following decision process:

1. All stage 0 decisions are made.
2. One scenario is selected at random, according to the scenario probabilities.
3. All stage 1 decisions are made.

The solution of a stochastic implosion problem is a fully-specified contingency plan for the above decision process. It specifies:

- The value of each stage 0 decision variable.
- For each scenario, s, the value that each stage 1 decision variable is to take, if (hypothetically) scenario s is randomly selected.

For each scenario, s, the stage 1 solution values for scenario s, together with the stage 0 solution values constitute the “scenario s solution”. The solution to a stochastic implosion problem is considered to be feasible, if and only if for each scenario, s, the scenario s solution is a feasible solution to the scenario s implosion problem. In other words, a feasible stochastic implosion solution represents a contingency plan that’s feasible with respect to all possible random outcomes.

For any particular scenario, s, the objective function for the scenario s implosion problem is exactly the same as the objective function for (deterministic) optimizing implosion for the scenario s implosion problem. Given a feasible solution to the stochastic implosion problem, the objective function value for stochastic implosion is defined to be the expected value of the objective function value of the scenario implosion solutions. Specifically, for each scenario, s:

- Let $\texttt{probability}_s$ be the probability of scenario s.
- Let $\texttt{objValue}_s$ be the value of the objective function for the scenario s implosion solution.

Then the objective function value of the stochastic implosion solution is:

$$\sum_s \texttt{probability}_s * \texttt{objValue}_s$$

where the sum is taken over all scenarios, s.

An optimal stochastic implosion solution is a feasible stochastic implosion solution that achieves the maximum possible value of this objective function among all feasible stochastic implosion solutions. In other words, it represents a contingency plan that achieves the best possible objective function value, in expectation. WIT's stochastic implosion facility computes an optimal stochastic implosion solution.

In general, some aspects of the data for a stochastic implosion problem will correspond to stage 0, some aspects will correspond to stage 1, and some will apply to both stages. WIT provides two alterative approaches to specifying this correspondence:

- In "stage-by-period" mode, the stage to which a data item belongs is determined by the period with which the data item is associated. For example the supplyVol of a part in period 2 might belong to stage 0, while the supplyVol of the part in period 3 belongs to stage 1. Any attribute that is not associated with a period (i.e., a scalar attribute) applies to both stages.
- In "stage-by-object" mode, the stage to which a data item belongs is determined by the object (e.g., part or operation) with which the data item is associated. For example the supplyVol of one part might belong to stage 0 in all periods, while the supplyVol of another part belongs to stage 1 in all periods. In this case, global attributes apply to both stages.

How to Invoke Stochastic Implosion

A typical WIT application program that uses stochastic implosion might do so by performing the following sequence of steps:

1. Build the "core" implosion problem.
   This is a deterministic implosion problem consisting of all aspects of the intended stochastic implosion problem that are not stochastic.
2. Set nScenarios.
   This is a global integer input attribute, ≥ 1, specifying the number of scenarios in the stochastic implosion problem.
3. Optionally set the stageByObject attribute.
   This is a global boolean input attribute:
   - If TRUE, stochastic implosion will proceed in stage-by-object mode.
   - If FALSE, stochastic implosion will proceed in stage-by-period mode.

   By default, its value is FALSE.
4. Set stochMode to TRUE.
   This is a global boolean input attribute. WIT is considered to be in "stochastic mode", if and only if this attribute is TRUE. When WIT is in stochastic mode, the following conditions hold:
   - The storage space for stochastic implosion is allocated.
   - Various attributes specific to stochastic implosion are now available.
   - The scenario-specific attributes can now be given different values in each scenario. At the beginning of stochastic mode, in each scenario, each

scenario-specific attribute is given the single value that the attribute had just before stochastic mode was entered.

- The structure of the problem is frozen: No data objects can be added or deleted.
- Many of the input attributes are frozen (essentially those that have no special significance to stochastic implosion).
- The function to solve the stochastic implosion problem is available.

5. Set the periodStage or objectStage attributes.
    - In stage-by-period mode, set the periodStage attribute.This is a global integer vector whose length is the number of periods. It is available only in stage-by-period mode. For each period, t, periodStage$_t$ identifies the stage to which period t belongs (i.e., 0 or 1). Thus for example, if periodStage = (0, 0, 0, 1, 1), then for each demand, the demandVol belongs to stage 0 in periods 0, 1, and 2 and belongs to stage 1 in periods 3 and 4.
    - In stage-by-object mode, set the objectStage attributes.This is an integer input attribute for parts and operations, available only in stage-by-object mode. For each part and operation, objectStage identifies the stage to which the part or operation belongs (i.e., 0 or 1). Each demand belongs to the same stage as its associated part; each BOM entry, substitute BOM entry, and BOP entry belongs to the same stage as its associated operation. All attributes of an object belong to the same stage as the object.
6. Loop through the set of all scenarios and perform steps 7-9.
    The scenarios are indexed from 0 to nScenarios - 1.
7. Set currentScenario.
    This is a global integer input attribute, available only in stochastic mode. Valid values: 0, ..., nScenarios - 1. Its value is the index of the "current scenario". In stochastic mode, for any given scenario-specific attribute, the corresponding API "`witSet`" and "`witGet`" functions store and retrieve the value of the attribute for the current scenario.
8. Set probability for the current scenario (if necessary).
    Probability is a scenario-specific global float input attribute, available only in stochastic mode. Valid values: 0.0 to 1.0. Its value represents the probability of the scenario. Initially, when stochMode is set to TRUE, the probability attribute for each scenario is given its default value of 1/nScenarios.
9. Set the scenario-specific attributes for the current scenario, as needed.
    - In stage-by-period mode, for each scenario-specific attribute and for each object that has this attribute, if the value of the attribute vector in the stage 1 periods is to be different in the current scenario from its value in the core problem, set the vector as appropriate.
    - In stage-by-object mode, for each scenario-specific attribute and for each stage 1 object that has this attribute, if the attribute value is to be different

in the current scenario from its value in the core problem, set it as appropriate.

10. Call `witStochImplode`.
This causes WIT to solve the stochastic implosion problem, by performing the following steps:
    - Perform error checking. (See the notes "When stochastic implosion is invoked" on page 28.)
    - Formulate the stochastic implosion problem as a linear programming problem.
    - Solve the linear programming problem by invoking CPLEX.
    - Form the stochastic implosion solution by translating from the solution to the linear programming problem.
11. Proceed in stochastic solution mode.
At the conclusion of `witStochImplode`, WIT sets the global boolean output attribute stochSolnMode to TRUE. WIT is considered to be in "stochastic solution mode", if and only if this attribute is TRUE. When WIT is in stochastic solution mode, the following conditions hold:
    - WIT is in a postprocessed state. (See "postprocessed" on page 161.)
    - The implosion solution attributes are now available.
    - Almost all input attributes are frozen.
12. In stage-by-object mode, retrieve the desired solution attributes of stage 0 objects.
While most solution attributes are scenario-specific, their values for stage 0 objects will be the same in all scenarios and therefore only need to be retrieved once.
13. Loop through the set of all scenarios and do steps 14-15.
14. Set currentScenario.
15. Retrieve the desired solution attributes for the current scenario.
    - In stage-by-object mode, this applies only to stage 1 objects.
    - In stage-by-period mode, this applies to all objects, but the values retrieved for stage 0 periods will be the same in all scenarios.

Data Attributes in Stochastic Implosion

Many of WIT's data attributes have special properties in the context of stochastic implosion. Specifically, each of following questions must be answered for each attribute:

1. Can the attribute be set and retrieved when stochMode is FALSE?
2. Can the attribute be set when stochMode is TRUE and stochSolnMode is FALSE?
3. Can the attribute be set when stochMode is TRUE and stochSolnMode is TRUE?
4. Is the attribute scenario-specific?

The answers to these questions are given in Table 2 on page 27.

NOTE: In this table, “As Normal” means "The same as without stochastic implosion".

TABLE 2 **Stochastic Implosion Aspects of WIT's Data Attributes**

Attribute	Object Type	Can set and retrieve when stochMode is FALSE?	Can set when stochMode is TRUE and stochSolnMode is FALSE?	Can set when stochMode is TRUE and stochSolnMode is TRUE?	Scenario-Specific?
nScenarios	Global	Yes	No	No	No
stageByObject					
stochMode	Global	Yes	Yes	Yes	No
stochSolnMode	Global	No	No	No	No
periodStage	Global	No	Yes	No	No
objectStage	Part				
	Operation				
currentScenario	Global	No	Yes	Yes	Yes
probability	Global	No	Yes	No	Yes
scrapCost	Part	Yes	Yes	No	Yes
stockBounds					
stockCost					
supplyVol					
cumShipBounds	Demand				
cumShipReward					
demandVol					
shipReward					
execBounds	Operation				
execCost					
subCost	Substitute				
consVol	Part	As Normal	No	No	Yes
excessVol					
prodVol					
residualVol					
scrapVol					
stockVol					
shipVol	Demand				
execVol	Operation				
subVol	Substitute				
appData	All	Yes	Yes	Yes	No
All Other Input Attributes		As Normal	No	No	No
All Other Output Attributes		As Normal			No
All Message Attributes					

NOTES:

1. When WIT enters stochastic mode (i.e., stochMode is changed from FALSE to TRUE), the following conditions are checked:
   - The problem must contain at least one part.
   - The computeCriticalList attribute must be FALSE.
   - The compPrices attribute must be FALSE.
   - The accAfterOptImp attribute must be FALSE.
   - The accAfterSoftLB attribute must be FALSE.
   - The mipMode attribute must be FALSE.
2. When WIT is in stochastic mode, the following API functions must not be called:
   - `witCopy<Object>Data`
   - `witEqHeurAlloc`
   - `witEqHeurAllocTwme`
   - `witEvalObjectives`
   - `witGeneratePriorities`
   - `witGetFocusShortageVol`
   - `witGetOperationFssExecVol`
   - `witGetPartFocusShortageVol`
   - `witGetSubsBomEntryFssSubVol`
   - `witHeurImplode`
   - `witMrp`
   - `witOptImplode`
   - `witPurgeData`
   - `witStartExtOpt`
   - `witStartHeurAlloc`
3. When setting the periodStage attribute, for each period t >0,
   if $\text{periodStage}_{t-1} = 1$, then periodStage_t must be equal to 1.
4. When stochastic implosion is invoked, the following conditions are checked:
   - stochMode must be TRUE.
   - stochSolnMode must be FALSE.
   - The sum of the probability attributes, over all scenarios, must be equal to 1.
5. When stochastic implosion is invoked in stage-by-period mode, the following additional condition is checked:
   - The value of each scenario-specific attribute of each object in each stage 0 period must be the same in all scenarios.
6. When stochastic implosion is invoked in stage-by-object mode, the following additional conditions are checked:

- The value of each scenario-specific attribute of each stage 0 object in each period must be the same in all scenarios.
- No BOM entry or substitute is allowed whose consuming operation belongs to stage 1 and whose consumed part belongs to stage 0.
- No BOP entry is allowed whose producing operation belongs to stage 1 and whose produced part belongs to stage 0.

Thus if an operation belongs to stage 1, then all parts that it connects to directly (through BOM entries, substitutes, and BOP entries) must also belong to stage 1. If an operation belongs to stage 0, then the parts that it connects to directly may belong to either stage.

7. To exit stochastic solution mode while keeping WIT in stochastic mode, call the API function `witClearStochSoln`. This function may only be called when stochMode and stochSolnMode are both TRUE. It puts WIT into an unpostprocessed state (See “postprocessed” on page 161.), sets the various solution attributes (e.g., execVol) to zero, and sets stochSolnMode to FALSE. At this point, WIT is in stochastic mode, but not stochastic solution mode. This allows the stochastic implosion problem to be updated and `witStochImplode` to be re-invoked, if appropriate.
8. A stochastic implosion problem can be specified in an ordinary WIT input data file: To do this, build a file that uses the ordinary “add” and “set” statements of a WIT input data file to perform steps 1 through 9 on page 23 - page 24.
9. If `witWriteData` is called when WIT is in stochastic mode, it writes an input data file that specifies the currently defined stochastic implosion problem.
10. If `witCopyData` is called when the source WitRun is in stochastic mode, the stochastic implosion problem stored in the source WitRun is copied into the destination WitRun.
11. Stochastic implosion generally takes much more CPU time and more memory than deterministic optimizing implosion.
12. Setting stochMode from FALSE to TRUE or TRUE to FALSE causes WIT to expend significant CPU time.
13. Setting the value of currentScenario causes WIT to expend significant CPU time. See “CPU-time efficiency” on page 189.
14. When using stochastic implosion, it is recomended that the optInitMethod attribute be set to crash. See “optInitMethod” on page 110.

PIP and Stochastic Implosion

The solution to a stochastic implosion problem can be pegged using post-implosion pegging. (See “Post-Implosion Pegging” on page 43.) If `witBuildPip` is called when WIT is in stochastic solution mode, a post-implosion pegging is constructed for the solution to the current scenario’s implosion problem. Whenever the value of currentScenario attribute is set, the PIP shipment sequence and the pegging are cleared. A complete pegging of the solution to a stochastic implosion problem can be obtained by looping through each of the scenarios and performing the following steps:

- Set currentScenario
- Specify the shipment sequence for the current scenario.
- Call `witBuildPip`.
- Retrieve the post-implosion pegging for the current scenario.

Note that the overall pegging that results from this procedure can be different in each scenario, even for periods and objects in stage 0. This makes sense because, for example, while the execution of a stage 0 operation must be the same in all scenarios, the use to which that execution is put may legitimately be different in each scenario.

Example: A Simple News Vendor Problem

To illustrate stochastic implosion, consider the following very simple example of the classic “news vendor problem”: A news vendor purchases a quantity of newspapers in the morning and then sells them throughout the day. There are 500 papers available to be purchased in the morning. The cost to buy the papers is $0.60 per paper and the revenue for selling them is $1.00 per paper. The vendor will generally sell as many papers as possible: no more than were purchased at the beginning of the day and no more than the demand over the course of the day. The difficulty is that when the vendor buys the papers, the demand is not known deterministically. For the simplicity of this example, we assume that the set of all possible demand volumes can be reduced to the following three representative cases:

- Low Demand: 200 papers, with probability = 25%
- Medium Demand: 300 papers, with probability = 50%
- High Demand: 400 papers, with probability = 25%

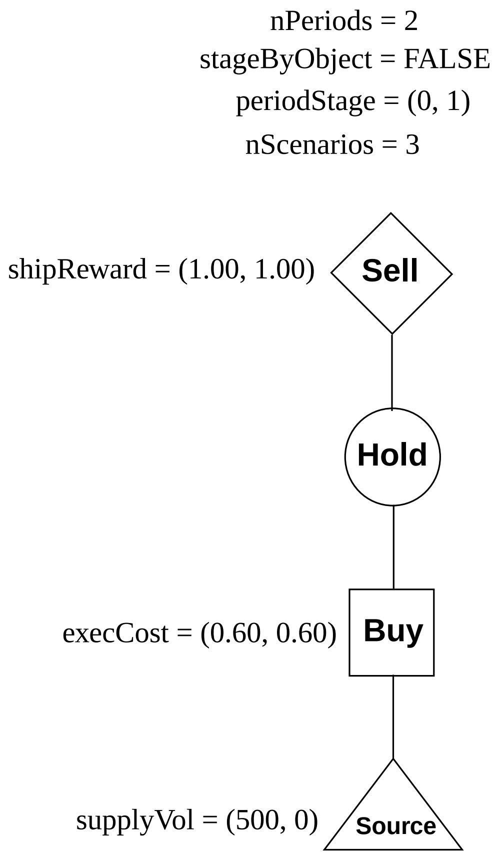
The problem is to construct the following contingency plan:

- How many papers to buy at the beginning of the day.
- How many papers to sell over the course of the day, given the demand.

The objective is to maximize the expected profit.

This problem may be modeled as a stochastic implosion problem using either stage-by-period mode or stage-by-object mode. In this example, stage-by-period mode will be used. The resulting stochastic implosion problem is shown in the following WIT diagram:

FIGURE 7 A Simple News Vendor Problem



The scenario-specific data for this problem is given in the following table:

TABLE 3 **Scenario-Specific Data for the News Vendor Example**

Scenario	Probability	DemandVol for Sell
0	0.25	(0, 200)
1	0.50	(0, 300)
2	0.25	(0, 400)

Invoking stochastic implosion on this problem results in the following solution:

- Buy.execVol = (300, 0)
- Scenario 0: Sell.shipVol = (0, 200)
- Scenario 1: Sell.shipVol = (0, 300)
- Scenario 2: Sell.shipVol = (0, 300)
- objValue = 95

So the contingency plan is to buy 300 papers, then if the demand is for 200 papers, sell 200 papers; if the demand is for 300 or 400 papers, sell 300 papers. The expected profit is $95.

An example WIT application program that uses stochastic implosion to solve this news vendor problem is given in Appendix A, page 433.

Additional Capabilities of WIT

Besides solving the implosion problem as described earlier in this chapter, WIT has a number of additional capabilities. These are:

- General:
  - Object Deletion
  - Object Iteration
  - Disallowing Late Shipments
  - User-Specified Solution
  - Testing the Feasibility of a User-Specified Solution
  - Critical Parts List
  - WIT-MRP
  - Focussed Shortage Schedule
  - Post-Implosion Pegging
- For Optimizing Implosion:
  - Evaluating the Objective Functions of a User-Specified Solution
  - User-Specified Starting Solution for Optimizing Implosion
  - Accelerated Optimizing Implosion
  - MIP Mode
  - External Optimizing Implosion
  - Disallowing Scrap
  - Multiple Objectives Mode
  - CPLEX Parameter Specifications
- For Heuristic Implosion:
  - Automatic Priority
  - Equitable Allocation Heuristic Implosion
  - Heuristic Allocation
  - Modifiable Heuristic Allocation
  - User-Specified Heuristic Starting Solution
  - Concurrent Pegging
  - Pegged Critical List
  - Multiple Routes
  - Proportionate Routing

- Build-ahead
- Stock Reallocation
- Multiple Execution Periods
- Single-Source
- Penalized Execution
- Penalized Execution with Proportionate Routing
- Selection Splitting
- Selection Recovery
- Lead Time Bounds

Object Deletion

This feature is available in API mode only.

WIT has six major types of non-global data objects:

- Parts
- Demands
- Operations
- BOM Entries
- Substitute BOM Entries
- BOP Entries

(The WIT problem itself is considered to be a “global” data object.)

In addition to allowing (non-global) data objects to be created and manipulated, WIT allows them to be individually deleted. This allows increased flexibility in building and modifying a WIT model. Deletion of non-global data objects is performed in two phases: selection and purge. In the selection phase, the user selects which data objects are to be deleted. In the purge phase, WIT deletes the selected objects. To select an object for deletion, simply set the object’s “selForDel” attribute to TRUE. (See Chapter 2.) To initiate a purge, invoke the API function `witPurgeData`. (See Chapter 5.)

Some types of objects have “prerequisites”. A prerequisite of an object, “A”, is another object, “B”, such that object “A” cannot exist unless object “B” exists. For example, a BOM entry cannot exist unless its consuming operation exists, so the consuming operation is a prerequisite for the BOM entry. Figure 8 and Table 4 show the prerequisite relationships between the different types of objects.

FIGURE 8 Prerequisite Relationships Between Data Object Types

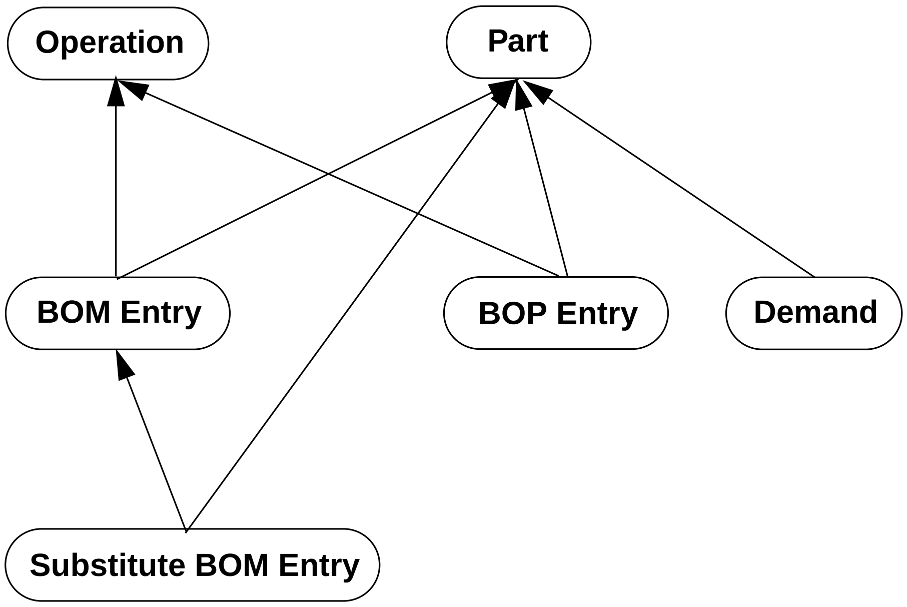


TABLE 4 Prerequisite Relationships Between Data Object Types

Data Object	Prerequisite	Prerequisite Relationship
BOM Entry	Operation	Consuming Operation
BOM Entry	Part	Consumed Part
Substitute BOM Entry	BOM Entry	Replaced BOM Entry
Substitute BOM Entry	Part	Consumed Part
BOP Entry	Operation	Producing Operation
BOP Entry	Part	Produced Part
Demand	Part	Demanded Part

During a purge, in addition to deleting the objects that were selected for deletion, WIT also deletes any data object that has one or more prerequisites that are being deleted. For example, if the user did not select a BOM entry for deletion, but did select its consuming operation, WIT will delete both the operation and the BOM entry during the purge.

Notice from Figure 8 that it is possible for an object to be a prerequisite of a prerequisite of an object. Such indirect prerequisite relationships will also cause WIT (during a purge) to delete objects that were not explicitly selected. For example, if the user did not select a substitute BOM entry for deletion or select its replaced BOM entry, but did select the consumed part for the replaced BOM entry, then WIT will delete the part, the replaced BOM entry and the substitute during the purge.

Finally, notice from Figure 8 that the prerequisite relationships never go more than two levels deep: There is no case of “a prerequisite of a prerequisite of a prerequisite”.

Object Iteration

The feature is available in API mode only.

WIT’s “object iteration” facility is a collection of API functions that enable the application program to traverse all of the (non-global) data objects of a WIT problem in the order in which they were created.

The object iteration process begins in an inactive state, advances through each of the data objects of the problem, and finally returns to the inactive state. Thus, at any point in time, the object iteration process for a WIT problem is considered to be in one of the following states:

- Inactive
- Located at a part
- Located at a demand
- Located at an operation
- Located at a BOM entry
- Located at a substitute
- Located at a BOP entry

The states other than “inactive” are all considered to be “active” states. The current state of the object iteration process is given by the global attribute “objItrState”.

The object iteration facility consists of the following API functions:

`witAdvanceObjItr`

This function advances the object iteration process. Its specific effect depends on the current state:

- If object iteration is currently inactive, it is advanced to the first data object in the problem.
- If object iteration is currently located at a data object other than the last data object in the problem, it is advanced to the next data object in the problem.
- If object iteration is currently located at the last data object in the problem, it is returned to its inactive state.

`witGetObjItrState`

This function retrieves the value of the objItrState attribute, which identifies the current state of the object iteration process.

`witGetObjItrPart`

If object iteration is currently located at a part, this function retrieves the identifying attribute for the part (i.e. the part's partName). If object iteration is not currently located at a part, a severe error is issued.

`witGetObjItrDemand`

`witGetObjItrOperation`

`witGetObjItrBomEntry`

`witGetObjItrSubsBomEntry`

`witGetObjItrBopEntry`

These functions are similar to `witGetObjItrPart`, but are to be used when object iteration is located at a demand, or an operation, etc.

`witResetObjItr`

This function returns the object iteration process to its inactive state.

NOTES:

1. When object iteration is active, the following actions will cause a severe error to be issued:
   - Creating a new data object in the same WIT problem, using `witAddPart`, `witReadData`, etc.
   - Requesting an object purge in the same WIT problem, using `witPurgeData`. (See "Object Deletion" on page 33.)
2. The following functions will put object iteration into its inactive state:
   - `witInitialize`
   - `witCopyData` (resets object iteration for the destination WIT problem)
3. The nominal use of object iteration is to traverse all of the data objects in the problem in the order in which they were created, as a heterogenous list. This might be used, for example, to implement a user-defined data file format.
4. Object iteration can also be used as a uniform way of traversing all of the data objects of a single type in a WIT problem, selecting the desired type by using the "objItrState" attribute. Thus a loop that traverses all of the substitute BOM entries of a WIT problem would look very similar to a loop that traverses all of the demands of a WIT problem. For an example of such a loop, see "Example of Application Code that Uses Object Iteration Functions" on page 344.

Disallowing Late Shipments

Ordinarily, late shipments are allowed in an implosion solution: demandVol in period t can be met by shipping in period t or in any period later than t. However, in some implosion applications, late shipments are unacceptable: The demand must be met on-time or not at all. To disallow late shipments to a specific demand, set the shipLateAllowed attribute to FALSE for the demand. (See “shipLateAllowed” on page 129.) The shipLateAllowed attribute is a boolean attribute for demands. Its default value is TRUE. When it is set to FALSE for a particular demand, the following constraint on the demand is added to the implosion problem:

For each period, t: $\text{shipVol}_t \leq \text{demandVol}_t$

This constraint is respected by both heuristic implosion and optimizing implosion.

In optimizing implosion mode, the shipLateAllowed attribute has an additional consequence: If shipLateAllowed is FALSE for a demand, then the LP problem generated by WIT in order to solve the optimizing implosion problem will not include any variables to measure the cumulative shipment volume for the demand. This results in a more compact LP problem. However, as a consequence of this, the following error conditions are imposed:

- For any demand, if shipLateAllowed = FALSE and cumShipReward $\neq$ 0, a severe error will be issued.
- For any demand, if shipLateAllowed = FALSE and cumShipBounds is not at its default set of values, a severe error will be issued.

User-Specified Solution

In some cases, it is useful for the user to specify a solution to the implosion problem and then have WIT perform various tasks with that sol7ution. Not all aspects of the implosion solution can be specified by the user; some aspects are always computed by WIT. However, the following attributes, which constitute the essential implosion solution, can be set by the user:

- execVol for each operation
- subVol for each substitute BOM entry
- shipVol for each demand

The user-specified solution can be used in the following four ways:

- Its feasibility can be tested: See “Testing the Feasibility of a User-Specified Solution” on page 38.
- The objective functions for it can be evaluated: See “Evaluating the Objective Functions of a User-Specified Solution” on page 54.
- It can be used as the starting solution for optimizing implosion: See “User-Specified Starting Solution for Optimizing Implosion” on page 54.

- It can be used as the starting solution for heuristic implosion: See "User-Specified Heuristic Starting Solution" on page 75.

Testing the Feasibility of a User-Specified Solution

One use of a user-specified solution is to have WIT determine whether or not it is feasible. That is, WIT can determine whether or not the user-specified solution satisfies all the constraints defined by the implosion problem. Feasibility is determined as part of WIT's postprocessing; for more details on postprocessing, see "postprocessed" on page 161. Postprocessing of a user-specified solution is invoked in API mode by calling the function, `witPostprocess.`

The user-specified solution are considered to be feasible, if and only if all of the following conditions are met:

- All shipment volumes are nonnegative.
- All execution volumes are nonnegative.
- All substitute volumes are nonnegative (subVols for each substitute BOM entry).
- For each demand stream in each period, the cumulative shipment volume is no greater than the cumulative demand volume.
- The total amount of substitution for a given BOM entry is no greater than the execution volume for the consuming operation.
- The execution volume for any operation is zero in any period in which execution is forbidden.
- The stocking and scrapping volumes implied by the solution are nonnegative.
- All hard lower and upper bounds are satisfied.
- The execution volumes of each operation are consistent with the operation's lot-size attributes: minLotSize, incLotSize, etc.

Critical Parts List

This capability is available both in heuristic and optimizing mode. If it has been requested, then heuristic and optimizing implosion will generate a "Critical Parts List" along with the implosion solution. WIT considers part `j` to be critical in period `t`, if it appears that increasing the supply of part `j` in period `t` would improve the solution. Critical parts and their periods are sorted in order of decreasing "urgency". How urgency of a part and period is defined depends on the implosion method:

- If an optimizing implosion was performed, the urgency of part `j` in period `t` is defined as the potential solution improvement per unit of additional supply volume of part `j` in period `t`.

- If a heuristic implosion was performed, the urgency of part `j` in period `t` is defined in terms of the priority of the demand that was gated by the supply of part `j` in period `t`. Specifically, consider both the earliest period in which some demand was not completely met and consider the highest priority demand that was not completely met in that period. This demand was not met because of a lack of sufficient supply of some part `j` in some period `t`. When the Critical Parts List is formed, part `j` and period `t` are placed at the top of the list. Then the part and period that caused the second highest priority demand not to be met in the earliest period is listed second in the list, and so on.

In general, the presence of part and period in the Critical Parts List should not be interpreted as a guarantee that increased supply of the part will result in an improved solution; rather, it should be interpreted more as a suggestion that the part is a good candidate for increased supply in that period.

Most of the parts that appear in the Critical Parts List are raw materials and capacities. However, a product may also appear in the Critical Parts List in any period early enough that the product can't be produced due to offsets.

Consider the following example:

A product is produced by an operation that has a BOM entry whose offset is 3. Then the product cannot be produced in periods 0, 1, or 2, and so the product may be listed as critical in periods 0, 1, or 2. If it listed as critical in these periods, interpret this to mean that there is "not enough" of this product being manufactured at the beginning of the time horizon. Expedited production or external supply of the product in these periods would likely lead to an improved solution.

See also "Pegged Criticial List" on page 78.

WIT-MRP

As an alternative to solving the implosion problem, WIT can optionally perform a limited, but very fast form of Material Requirements Planning, called "WIT-MRP". The WIT-MRP takes the complete BOM structure and performs an explosion to get a Requirements Schedule. Other MRP systems do this but they take more factors into account than WIT-MRP does.

WIT-MRP uses most of the information, such as yield rate, fallout rate, effectivity dates found in the WIT BOM. WIT-MRP does not explode through substitute BOM entries; however, it may use available supplies of the part consumed by a substitute. (See "netAllowed" on page 146.) In some cases, a demand cannot be propagated all the way to raw materials, because the offsets imply consumption in negative periods. In such a case, WIT-MRP puts requirements on products.

Alternatively, WIT-MRP can be used with “truncated offsets”. This causes offsets that normally imply consumption in negative periods to be re-interpreted to imply consumption in period 0. This allows WIT-MRP to propagate requirements all the way to the bottom of the complete BOM structure, but results in an infeasible implied production schedule. For more information on truncated offsets, see “truncOffsets” on page 115.

In general, MRP is well-defined for single method assembly problems and on these problems, WIT-MRP produces a very appropriate requirement schedule. On non-SMA problems, WIT-MRP may have difficulties. If an operation has more than one BOP entry in its BOP, then when WIT-MRP executes the operation in order to produce one of the produced parts, the volumes of the other parts produced may be wasted. If a part appears as the produced part of more than one BOP entry, WIT-MRP will choose one non-by-product BOP entry and, whenever more of the part is needed, it will execute the producing operation for that BOP entry only.

Focussed Shortage Schedule

Focussed Shortage Schedule (FSS) is an optional feature available in both heuristic and optimizing implosion modes. Performed after an implosion, it provides the user with guidance as to what supplies need to be increased to better meet the demands.

There are actually two ways to use FSS: “focus mode” and “schedule mode”. We will explain focus mode first.

To use FSS in focus mode, for each demand, i, you specify a focusHorizon_i, where $-1 \le \text{focusHorizon}_i \le \text{nPeriods} - 1$. Setting focusHorizon_i to some value ≥ 0 means that this demand is considered to be so important that you would be willing to increase supply volumes as necessary in order to meet this demand in periods 0, 1, ..., focusHorizon_i. Setting $\text{focusHorizon}_i = -1$ means that you are not willing to increase supplies to meet this demand. The set of all demands i such that $\text{focusHorizon}_i \ge 0$ along with the corresponding values of focusHorizon_i is called the “focus” of the FSS.

In focus mode, WIT uses the focusHorizons to compute the “FSS Shipment Schedule”, fssShipVol, as follows:

- For each demand i, for each period $t \le \text{focusHorizon}_i$, ship all of the demand in period t on time.
- For each demand i, for each period $t > \text{focusHorizon}_i$, ship according to a shipment schedule whose backlog is no larger than the corresponding backlog in the current implosion shipment schedule.

Once the FSS shipment schedule has been determined, WIT computes the Focussed Shortage Schedule itself. For each part j and period t, the Focussed Shortage Schedule specifies $\text{focusShortageVol}_{j,t}$, which is an amount such that, if the supply of part `j` in period `t` were increased by focusShortageVol$_{j,t}$ for all parts and periods, it would be possible to meet the FSS shipment schedule.

In schedule mode, you don't specify the focusHorizons; you just specify the FSS shipment schedule directly. This is much more flexible, but somewhat more sophisticated to use than focus mode.

To distinguish between focus mode and schedule mode, WIT provides a global attribute, "useFocusHorizons". To use focus mode, you set useFocusHorizons to TRUE; to use schedule mode, you set useFocusHorizons to FALSE. Thus in focus mode, you use FSS as follows:

- Set useFocusHorizons to TRUE.
- Specify the focusHorizons.
- Invoke FSS. At this point, WIT will compute the FSS shipment schedule and then the Focussed Shortage Schedule corresponding to it.

In schedule mode, you use FSS as follows:

- Set useFocusHorizons to FALSE
- Specify the FSS shipment schedule.
- Invoke FSS. WIT will compute the Focussed Shortage Schedule corresponding to your FSS shipment schedule.

In API mode, FSS is computed on an "as needed" basis. Whenever an API function is called that returns one of the outputs of FSS, if the FSS has not yet been computed relative to the current implosion solution, WIT automatically computes FSS at that time. This applies to the following API functions:

- `witGetFocusShortageVol`
- `witGetPartFocusShortageVol`
- `witGetOperationFssExecVol`
- `witGetSubsBomEntryFssSubVol`

If any of these functions is called when the FSS has not yet been computed for the current implosion solution, WIT will compute it automatically. Thus if the application program calls one of these functions when the FSS has not yet been computed, the program is considered to be "invoking FSS".

The Stand-Alone Executable version of WIT only allows focus mode. Furthermore, in Stand-Alone mode, the only focus allowed is the universal focus, i.e, focusHorizon$_{i}$ = nPeriods - 1. The FSS with universal focus is computed and printed if the Comprehensive Implosion Solution Output File is requested in the Control Parameter file. This restriction on the use of FSS does not apply in API mode, which allows completely general use of FSS.

NOTES:

1. WIT must be in a postprocessed state when FSS is invoked in API mode. If it is not, a severe error is issued. See “postprocessed” on page 161. In stand-alone mode, this issue is handled automatically.
2. Although the FSS shipment schedule is guaranteed to be feasible if the FSS is added to the supply schedule, it might not be the shipment schedule that the optimizing or heuristic implosion would find. In the case of optimizing implosion, WIT might use the supply to find an even “better” shipment schedule, according to the objective function, or perhaps just a different “equally good” schedule. In the case of heuristic implosion, the schedule that WIT finds may be better, equally good, or worse than the FSS shipment schedule. This is simply a consequence of using a heuristic instead of optimization.
3. The FSS might not be “minimal”, i.e, it might be possible to meet the FSS shipment schedule with even less supplies than indicated by the FSS.
4. The FSS computation ignores upper bounds on operation execution. However, the effect of upper bounds on execution can always be achieved by constraining execution with capacity. FSS does respect capacity.
5. FSS ignores upper and lower bounds on cumulative shipment, because cumulative shipment is completely determined by the FSS shipment schedule.
6. FSS respects upper and lower bounds on stock and lower bounds on execution to the extent that they are respected by the current implosion solution. Thus if the current implosion solution satisfies a bound, that bound will also be satisfied by the FSS; if the current implosion solution violates a bound by x units, the FSS may also violate the bound by x units.
7. The FSS is computed by invoking WIT-MRP to determine what additional supplies are needed to meet those aspects of the FSS shipment schedule that are not already met by the current implosion solution. Because the FSS is computed using MRP and similar methods, it is very fast, much faster than a heuristic implosion. It is well suited for iterative use.
8. FSS works best with single method assembly problems. FSS does give correct answers for non-SMA problems, but since the FSS computation depends on WIT-MRP (which has difficulties with non-SMA problems) the resulting schedule in this case might be far from minimal.
9. FSS cannot be used with a user-specified solution. (See “User-Specified Solution” on page 37.) The solution must be a direct result of heuristic or optimizing implosion.
10. The useFocusHorizons attribute defaults to TRUE.
11. The default value of focusHorizon is - 1 (the empty focus).
12. In schedule mode, the FSS shipment schedule defaults to the implosion shipment schedule. It is set to the implosion shipment schedule whenever an implosion is performed. Thus you should only specify the FSS shipment schedule after implosion, because if you set it before implosion, it will be overwritten.

One good way to use FSS in focus mode is as follows:

1. Set useFocusHorizons to TRUE.
2. Do an implosion (heuristic or optimizing).
3. Specify the universal focus focusHorizon$_i$ = nPeriods -1 for all demands, i.
4. Examine the resulting FSS. When the universal focus is specified, the FSS is simply a schedule of additional supply volumes sufficient to meet all of the demands, on time, in all periods.
5. It may not be possible to obtain all of the additional supply volumes indicated by the FSS that corresponds to the universal focus. If this is the case, do a second iteration and “narrow” the focus, i.e., set focusHorizon$_i$ to some earlier period, for some of the demands. In particular, you might set focusHorizon$_i$ = -1 for some of the demands, thereby removing them from the focus altogether. After you have altered (narrowed) the focus, invoke FSS.
6. Repeat this process as often as necessary. Adjust the focus and compute the resulting shortage schedule, until you converge on a Focussed Shortage Schedule that you can achieve (from your suppliers) and a FSS shipment schedule you can accept.

In general, using FSS in schedule mode allows you much more flexibility than in focus mode:

Consider the following example: Suppose the implosion solution has met one of the demands two periods late. By using FSS in focus mode, you have determined what additional supplies would be sufficient to meet the demand on time and have determined that these additional supplies are unattainable. So now you want to know what additional supplies are needed to meet the demand one period late. There is no way to determine this in focus mode. In schedule mode, you can determine this by setting the FSS shipment schedule to the implosion shipment schedule, replacing the two-period-late shipment by a one-period-late shipment, and then invoking FSS.

Post-Implosion Pegging

A “pegging” of an implosion solution considers each resource allocated in the solution and designates a set of shipments to which that resource is considered to have been allocated. Note that this designation is necessarily somewhat arbitrary, since heuristic implosion and optimizing implosion do not explicitly allocate resources to specific shipments; they merely generate a feasible solution to the implosion problem (guided by the priorities, objective functions, etc.). WIT provides two general kinds of pegging: “post-implosion pegging” (described in this section) and “concurrent pegging” (described in a later section).

The “post-implosion pegging” techniques (PIP) can be applied to any implosion solution: heuristic implosion/allocation, optimizing implosion, or even a user-specified solution. The pegging is formed by reconstructing the solution after it was computed and pegging the reconstruction process.

PIP provides pegging for the following attributes:

- operation.execVol
- subEntry.subVol
- part.supplyVol
- part.prodVol
- part.consVol

The PIP technique requires two inputs:

- A feasible implosion solution.
- A “shipment sequence” for the implosion solution (see below).

To define the concept of a shipment sequence, consider any ordered list of shipment triples (demand, period, incShipVol). For any demand and period, define the sequenced shipVol to be the sum of the incShipVols over all triples in the sequence that match that demand and period. Then an ordered list of shipment triples is considered to be a shipment sequence for an implosion solution, if for any demand and period, its sequenced shipVol is no larger than the corresponding shipVol in the implosion solution. In effect, a shipment sequence takes the shipment schedule of an implosion solution (or a portion of it) and makes it sequential.

The PIP algorithm uses the shipment sequence in order to build the pegging as follows: The triples in the sequence are considered in order. For each triple, the algorithm constructs a corresponding partial solution that uses no resources that were pegged to a previous triple and then pegs the partial solution’s resources to the demand and period of the current triple.

The general way to specify the shipment sequence is by invoking the following two API functions:

- `witClearPipSeq` which clears the shipment sequence.
- `witAppendToPipSeq` which appends a triple to the end of the shipment sequence.

These two functions can be called either before implosion or between implosion invoking PIP.

As an alternative, when the heuristic is being used, the shipment sequence can be specified by setting the following global boolean attribute to TRUE, before invoking the heuristic:

- pipSeqFromHeur

(See “pipSeqFromHeur” on page 112.) When this boolean is TRUE, the heuristic sets the shipment sequence to be the sequence of shipment triples that it used to construct the solution. When using PIP with the heuristic, this may often be an appropriate shipment sequence to use.

To have WIT perform post-implosion pegging, invoke the following API function:

- `witBuildPip`

To retrieve the post-implosion execVol pegging, call `witGetDemandExecVolPip.` When calling this function, the application program specifies a demand and shipment period. The function returns several lists that constitute a list of “pegging triples”. Each pegging triple specifies an operation, an execution period and a “pegged execVol”. The pegged execVol is the portion of the execVol of the operation in the execution period that is pegged to the demand in the shipment period.

To retrieve the post-implosion pegging for subVol, supplyVol, prodVol, or consVol, call the following functions, which work similarly to `witGetDemandExecVolPip`:

- `witGetDemandSubVolPip`
- `witGetDemandSupplyVolPip`
- `witGetDemandProdVolPip`
- `witGetDemandConsVolPip`

<u>PIP in the Absence Side-Effects</u>

To understand the nature of the pegging produced by PIP, consider an implosion problem that satisfies the following restrictions:

- No operation has more than one BOP entry.
- Each BOP entry has expAllowed = true.
- If a BOP entry has a positive prodRate, it is not so small that WIT can’t use it for explosion (See “expCutoff” on page 102.).
- No consRate is < 0.

A problem that satisfies these restrictions can be called a “no side-effects implosion problem”. A problem that does not satisfy these restrictions is considered to have side-effects. When PIP is used on a no side-effects implosion problem, the pegging that it produces is actually a “feasible partitioning” of the implosion solution.This means that it has the following two properties:

1. The sum of the pegged execVols for a given operation and period is ≤ the corresponding execVol in the solution. The same holds for subVols, prodVols, and consVols. Finally, the sum of the pegged supplyVols for a

given operation and period is ≤ the corresponding supplyVol - excessVol in the solution.

2. Given any demand and ship period, define the “pegged implosion problem” for that demand and period to be the original implosion problem with the supplyVols replaced by the supplyVols pegged to that demand and ship period. All lot sizes are set to 0. Define the “pegged implosion solution” for the demand and ship period to be the pegged execVols, subVols, prodVols, and consVols for that demand and ship period, along with the actual shipVol for that demand and ship period. All other shipVols, execVols, subVols, prodVols, and consVols are 0. Then the pegged implosion solution for any demand and ship period will be a feasible solution to the corresponding pegged implosion problem. Also, the scrapVols in this feasible solution are all zero.

This is a very desirable property for a pegging. Unfortunately, a feasible partitioning is often (or usually) impossible for problems that have side-effects.

<u>PIP with Side-Effects</u>

To understand how PIP works on problems with side-effects, we now consider the class of problems in which no operation has multiple explodeable BOP entries (i.e., more than operation with expAllowed = TRUE).

To handle side-effects, the PIP algorithm works with the following auxiliary problem: A BOP entry that’s ineligible for explosion is called a side-effect BOP entry; a BOM entry with a negative consRate is called a side-effect BOM entry. For each part and each period, a quantity called “sideVol” is computed. This is the total production (prodVol) of the part due to side-effect BOP entries minus the total consumption (consVol) of the part due to side-effect BOM entries. (This is a positive number minus a negative number.) The sideVol of each part is then added to its supplyVol. Finally, all side-effect BOP entries and BOM entries are removed from the problem. This auxiliary problem has the following properties:

- It is a no-side-effects problem.
- The solution to the original problem is also a feasible solution to the auxiliary problem.

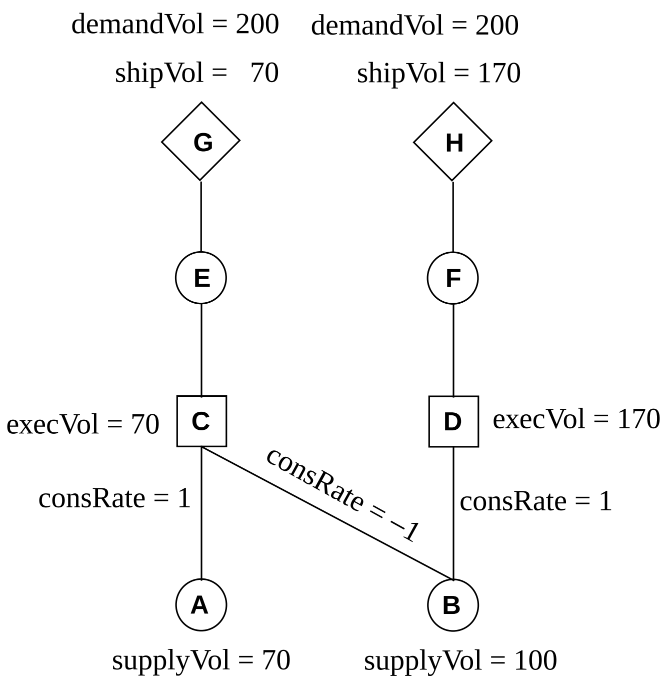
The pegging produced by the PIP algorithm is a feasible partitioning of the original solution with respect to the auxiliary problem.

For reporting purposes, the pegged supplyVol of a part and its pegged sideVol are kept separate. Thus when `witGetDemandSupplyVolPip` is called, this retrieves the pegging of the actual supplyVols, not the sideVols. To retrieve the sideVol pegging, call the following functrion:

- `witGetDemandSideVolPip`

Consider the following example:

FIGURE 9 PIP with Side-Effects



(where nPeriods = 1) Here, the BOM entry from operation C to part B is a side-effect BOM entry. The total consumption of part B due to side-effect BOM entries is –70, so the sideVol part B is 70. The supplyVol and sideVol pegging is as follows:

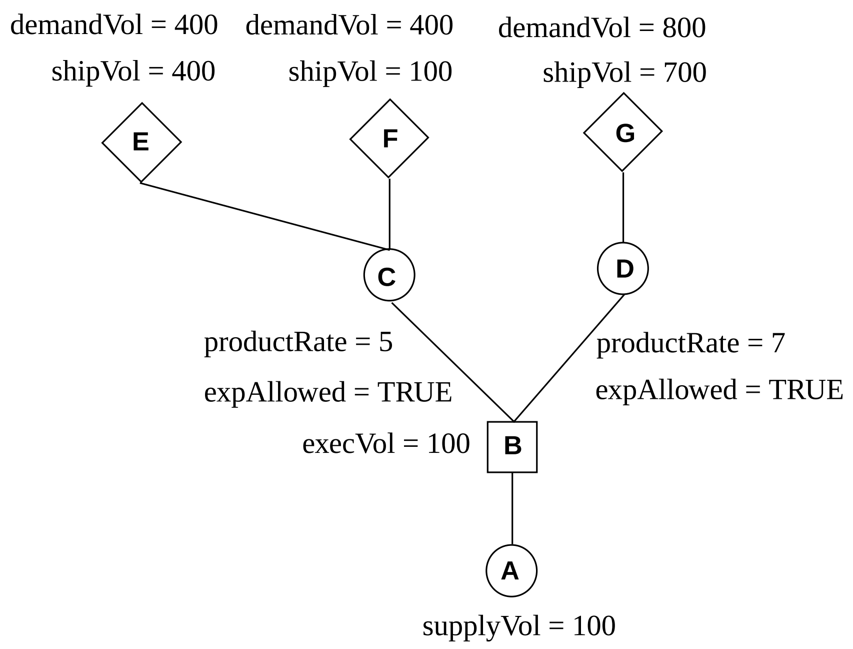
- 70 units of supplyVol for part A are pegged to demand G.
- 100 units of supplyVol for part B are pegged to demand H.
- 70 units of sideVol for part B are pegged to demand H.

PIP with Multiple Explodeable BOP Entries

Finally, consider problems in which some operations have multiple explodeable BOP entries. When an operation has multiple explodeable BOP entries, PIP pegs the entire execVol for the operation through each of the operation's explodeable BOP entries. As a result, each unit of execVol may be pegged jointly to more than one demand. This is called a "group" pegging. This group pegging is propagated downward in the BOM structure to all of the supplyVols, subVols, etc. that PIP identifies as having been used in order to make the execVol feasible.

Consider the following example:

FIGURE 10 PIP with Multiple Explodeable BOP Entries



(where nPeriods = 1)

When PIP is applied to this problem, the following execVol pegging results:

- 80 units of execVol for operation B are pegged jointly to demands E and G.
- 20 units of execVol for operation B are pegged jointly to demands F and G.

When the pegging is retrieved through the API, the pegging for each individual demand is given:

- 80 units of execVol for operation B are pegged to demand E.
- 20 units of execVol for operation B are pegged to demand F.
- 100 units of execVol for operation B are pegged to demand G.

In addition to the main attributes for which PIP provides peggings (execVol, supplyVol, etc.), there is one additional pegged quantity computed by PIP that is relevant specifically when PIP is applied to problems that contain operations with multiple explodeable BOP entries: “coExecVol” (co-execution volume) for BOP entries. For any BOP entry of an operation, any execution period, any demand, and any shipment period, the corresponding pegged coExecVol is the portion of the execVol of the operation in the execution period that’s pegged to the demand in the shipment period and whose pegging is specifically associated with the BOP entry. This has the effect of separating the joint pegging of the operation's execVol into smaller groups of shipments (often individual shipments) associated with each of the operation's explodeable BOP entries.

In the example given above, PIP constructs the following coExecVol pegging:

- 80 units of coExecVol for BOP entry "B to C" are pegged to demand E.
- 20 units of coExecVol for BOP entry "B to C" are pegged to demand F.
- 100 units of coExecVol for BOP entry "B to D" are pegged to demand G.

Note that the coExecVol pegging is also available in problems that have no operations with multiple explodeable BOP entries, but in this case, the coExecVol pegging is completely equivalant to the execVol pegging.

<u>Pegging Order</u>

PIP's "pegging order" is the order in which it pegs the supplies, production, and side-effects of a material part, considered over all periods. By default, PIP uses the NSTN (No-Sooner-Than-Necessary) pegging order, which can be described as follows:

- First, the supply of the part is pegged in periods nPeriods-1, ..., 1, 0.
- Next, the production of the part is pegged in periods nPeriods-1, ..., 1, 0.
- Lastly, the sideVol for the part is pegged in periods nPeriods-1, ..., 1, 0.

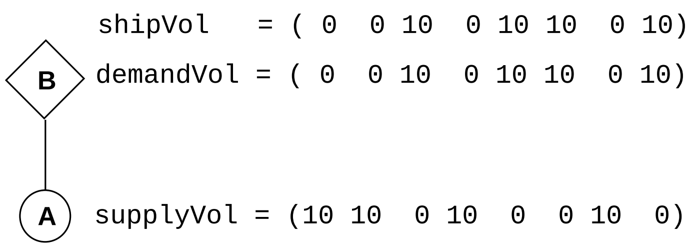
Alternatively, the ASAP (As-Soon-As-Possible) pegging order can be used, which can be described as follows:

- First, the supply of the part is pegged in periods 0, 1, ..., nPeriods-1.
- Next, the production of the part is pegged in periods 0, 1, ..., nPeriods-1.
- Lastly, the sideVol for the part is pegged in periods 0, 1, ..., nPeriods-1.

To have PIP use the ASAP pegging order on a material part, set the asapPipOrder attribute for the part to TRUE. See "asapPipOrder" on page 119.

Consider the following example:

FIGURE 11 PIP Pegging Order


shipVol   = (0 0 10 0 10 10 0 10)
B
demandVol = (0 0 10 0 10 10 0 10)
A
supplyVol = (10 10 0 10 0 0 10 0)

Suppose heuristic implosion is used, with pipSeqFromHeur = TRUE. If the NSTN pegging order is used (`C.asapPipOrder = FALSE`), then PIP will construct the following pegging:

- 10 units of A.supplyVol in period 1 are pegged to demand B in period 2.
- 10 units of A.supplyVol in period 3 are pegged to demand B in period 4.
- 10 units of A.supplyVol in period 0 are pegged to demand B in period 5.
- 10 units of A.supplyVol in period 6 are pegged to demand B in period 7.

In contrast, if the ASAP pegging order is used (`C.asapPipOrder = TRUE`), then PIP will construct the following pegging:

- 10 units of A.supplyVol in period 0 are pegged to demand B in period 2.
- 10 units of A.supplyVol in period 1 are pegged to demand B in period 4.
- 10 units of A.supplyVol in period 3 are pegged to demand B in period 5.
- 10 units of A.supplyVol in period 6 are pegged to demand B in period 7.

In this case, the pegging produced by the ASAP pegging order may be considered more “orderly” than the one produced by the NSTN pegging order.

In general, the NSTN pegging order approximates heuristic implosion’s NSTN build-ahead technique; the ASAP pegging order approximates the heuristic’s ASAP build-ahead technique and seems to produce more orderly peggings: its behavior resembles that of a FIFO queue.

PIP to Operations

The “PIP to operations” capability considers each resource allocated in the solution and designates a set of operations and periods to which that resource is considered to have been allocated. It provides an answer to the question: "Which resources were used to enable a particular operation to be executed in a particular period?"

To have WIT perform PIP to operations, you do the following steps:

- Start with an implosion solution, as usual.
- Specify a shipment sequence, as usual.
- Specify some operations as “PIP enabled”. You do this by setting the pipEnabled attribute of the operation to TRUE. See “pipEnabled” on page 136.
- Invoke `witBuildPip,` as usual.

This will cause WIT to compute the PIP to demands and the PIP to operations together. Operation peggings will be computed only to PIP enabled operations.

To retrieve the post-implosion pegging to an operation, invoke any of the following functions:

- `witGetOperationSupplyVolPip`
- `witGetOperationProdVolPip`
- `witGetOperationConsVolPip`
- `witGetOperationExecVolPip`
- `witGetOperationSubVolPip`
- `witGetOperationCoExecVolPip`
- `witGetOperationSideVolPip`

These functions work the same way as their demand counterparts, except that you specify an operation instead of a demand. Also note the following:

- The PIP to operations retrieval functions must not be invoked on operations that are not PIP enabled.
- The execVol of an operation will not be pegged to the same operation.

You can exercise some control over the pegging to operations process by setting the pipRank attribute. (See “pipRank” on page 136.) This is an int attribute of operations. The PIP algorithm uses it in the following way: When the PIP algorithm pegs a quantity of an attribute (e.g. supplyVol) to a demand shipment, it simultaneously pegs the same quantity to all of the PIP enabled operations that contributed to the shipment by using the pegged quantity. The demand and the PIP enabled operations are called the recipient list of that particular pegging. Often, the algorithm must choose between two or more potential recipient lists to which a quantity is to be pegged. When this happens, it uses the following logic:

1. If two recipient lists contain different elements of the shipment sequence, the list with the earlier shipment sequence element is chosen. This is the case that probably happens most of the time.
2. If two recipient lists contain the same element of the shipment sequence, but the one recipient list contains an operation whose pipRank is higher than the highest pipRank in the other list, the higher pipRank list is chosen.
3. If two recipient lists contain the same element of the shipment sequence, and the highest pipRanks in the two lists are equal, a tie-breaking rule is applied (too complicated to explain here).

Note that the pipRank attribute can be left at its default value, which is 0. If this is done, then case 2 never happens and the algorithm just uses case 1 (most of the time) and case 3 (tie-breaker), which may be perfectly acceptable.

The peggings for PIP to operations have the following consistency properties:

- They are flow consistent: Each quantity of aa pegged attribute (e.g. supplyVol) will show up in the pegging to every PIP-enabled operation that it “passed through” on its way to the demand to which is it ultimately pegged.

- They are demand consistent: If you run PIP twice with the same shipment sequence, but with differing sets of PIP-enabled operations and/or differing pipRanks, the peggings to the demands will be the same in both peggings.
- They are high-rank consistent:
  - Suppose you run PIP twice.
  - Let A be the set of PIP enabled operations the first time.
  - Let B be the set of PIP enabled operations the second time
  - Suppose B is a superset of A.
  - Suppose the pipRanks of the A operations are the same in both runs.
  - Suppose the pipRank of every operation in A is strictly greater than the pipRank of every operation in B but not in A.
  - Then the peggings to the operations in A will be the same in both peggings.

The demand consistency and high-rank consistency properties are good to know about: They tell you that you can peg to whatever subset of operations you want (by making them PIP-enabled) with the assurance that (hypothetically) you could peg to all operations, giving the newly PIP-enabled operations lower pipRank, and the resulting pegging would be consistent with the pegging you are looking at.

NOTES:

1. The following API functions pertain to PIP:
   - `witClearPipSeq`
   - `witAppendToPipSeq`
   - `witSetPipSeqFromHeur`
   - `witGetPipSeqFromHeur`
   - `witGetPipSeq`
   - `witSetPartAsapPipOrder`
   - `witGetPartAsapPipOrder`
   - `witSetOperationPipEnabled`
   - `witGetOperationPipEnabled`
   - `witSetOperationPipRank`
   - `witGetOperationPipRank`
   - `witBuildPip`
   - `witGetPipExists`
   - `witGetDemandExecVolPip`
   - `witGetDemandSubVolPip`
   - `witGetDemandSupplyVolPip`
   - `witGetDemandProdVolPip`
   - `witGetDemandConsVolPip`
   - `witGetDemandSideVolPip`
   - `witGetDemandCoExecVolPip`
   - `witGetOperationExecVolPip`
   - `witGetOperationSubVolPip`

- `witGetOperationSupplyVolPip`
- `witGetOperationProdVolPip`
- `witGetOperationConsVolPip`
- `witGetOperationSideVolPip`
- `witGetOperationCoExecVolPip`

2. PIP ignores lot sizes. This allows resources that were used together for a lot size constraint to be pegged to different shipments.
3. When `witBuildPip` is invoked, if the shipment sequence defines any sequenced shipVols that are larger the corresponding shipVols in the implosion solution, the incShipVols in the shipment sequence are automatically reduced as necessary.
4. The WitRun must be in a postprocessed state when `witBuildPip` is invoked. (See “postprocessed” on page 161.)
5. If PIP is to be invoked, the problem must not have any substitute BOM entry with a negative consRate.
6. The global boolean attribute “pipExists” indicates whether or not a post-implosion pegging has been computed and is currently available to be retrieved. (See “pipExists” on page 112.) Upon completion of `witBuildPip`, pipExists is set to TRUE. Any action that takes the WitRun out of a postprocessed state sets pipExists to FALSE. The pipExists attribute must be TRUE, when any of the PIP retrieving functions is called, e.g., `witGetDemandExecVolPip`. Also, `witClearPipSeq` and `witAppendToPipSeq` set pipExists to FALSE.
7. When applied to problems in which contain no PIP enabled operations and no operations with multiple explodeable BOP entries, PIP can be expected to take a modest amount of run time and memory (bounded by a low-order polynomial in the worst case).
8. When applied to problems that contain either PIP enabled operations or operations with multiple explodeable BOP entries, PIP can take an exponential run time and exponential memory in the worst case. Fortunately, the exponential behavior is reasonably well-contained. Let D = the “multiple pegging depth” of an implosion problem. This is the maximum number of operations that are either PIP enabled or have multiple explodeable BOP entries that can be visited by traversing the complete BOM structure from top to bottom. In other words, D is the number of distinct levels in the BOM structure at which PIP enabled operations or multiple explodeable BOP entries occur. The run time and memory requirement of PIP is bounded by a function that is exponential in D. So for example, if all the PIP enabled operations occur at one level and no operation has multiple explodeable BOP entries occur, then D = 1, and the run time and memory will probably be reasonable.

   Also note that the growth is exponential in the worst case: It’s possible to create WIT problems in which all operations are PIP enabled, resulting a large value of D, but the run time and memory usage of PIP are still not large. The only way to know is to try it.

Evaluating the Objective Functions of a User-Specified Solution

The concept if a user-specified solution was introduced on page 37. Another use of a user-specified solution is to have WIT compute the value of the objective functions corresponding to it. The values computed are objValue and boundsValue.

This feature is available in API mode by calling the function, `witEvalObjectives`. See “Action Functions” on page 276.

User-Specified Starting Solution for Optimizing Implosion

Normally, optimizing implosion has its own way of computing an initial solution. However, as an alternative, it can use a user-specified solution as its starting solution. (See page 37 for the definition of a user-specified solution.) To have optimizing implosion use the user-specified solution as its starting solution, set the optInitMethod attribute to “schedule”. (See “optInitMethod” on page 110.) Or just set the value of any of the attributes execVol, subVol, or shipVol.

If the user-specified solution was produced by a previous run of optimizing implosion for a very similar data set (differing only in, e.g., a few supply quantities or demand quantities), significant speed ups can sometimes be achieved, relative to running optimizing implosion without a a user-specifed starting solution. The purpose of this feature is to allow optimizing implosion to be used effectively in an iterative mode: One would run optimizing implosion initially without a user-specified solution, look at the result, modify the data slightly (e.g., by increasing a supply), re-run optimizing implosion using the previous solution, and keep repeating this until a satisfactory result has been achieved.

Unfortunately, although the purpose the user-specified starting solution is to speed up optimizing implosion, we can't guarantee that using this feature will actually result in a substantial speed-up in all cases. In our tests, running optimizing implosion with a good initial solution resulted in runs that were 2 to 3 times faster than without a user-specified solution for some problems, but only 10-20% faster for other problems. Our advice is to try the user-specified starting solution feature on typical data sets that you use, and then only continue to use this feature, if you find that it improves your run times. As an alternative, see “Accelerated Optimizing Implosion” on page 55.

The user-specified solution does not have to feasible for this capability to work, although it is likely to work better if the solution is feasible.

Accelerated Optimizing Implosion

This feature is available only in optimizing mode and only by using the API. It is not available in stand-alone mode.

In many applications, implosion is used repetitively: The user defines an implosion problem to WIT, has WIT perform an implosion, and then, based on the output, changes the input data, has WIT perform another implosion, and repeats this process many times. In many cases, when optimizing implosion is being used repetitively, all but the first implosion can made to run much faster than first implosion, by using accelerated optimizing implosion.

Accelerated optimizing implosion works as follows: At the end of the first optimizing implosion, the LP model, LP solution, and CPLEX environment are kept in memory. Then, when the second optimizing implosion is requested, the LP model and CPLEX environment are updated to coincide with the new implosion problem and the LP solution from the first optimizing implosion is used as the starting solution for the second optimizing implosion. In tests, we found that this approach runs much faster than ordinary optimizing implosion, perhaps as much as 10 times faster.

There are two restrictions that apply to the use of accelerated optimizing implosion. First, as described above, accelerated optimizing implosion works by keeping various information in memory. As a consequence, it is necessary to do multiple optimizing implosions in the same process (the same run of the program), altering the input data between the implosions.

The second restriction is that only certain changes to WIT's input data are compatible with accelerated optimizing implosion. At the end of the first optimizing implosion, if accelerated optimizing implosion has been requested, WIT is considered to be in an “accelerated state”, meaning that the LP data has been kept in memory. There is a selected subset of WIT's input data attributes for which changes are “compatible” with an accelerated state, i.e., if the value of any of these attributes is changed, accelerated mode is preserved. If the value of any other input data attribute is changed, WIT goes immediately into an unaccelerated state. The next time an accelerated optimizing implosion is requested, if WIT is in an accelerated state, it will perform an accelerated optimizing implosion; if it is in an unaccelerated state, it will perform an ordinary (non-accelerated) optimizing implosion. (Roughly speaking, WIT goes into an unaccelerated state, whenever a change is made to the input data that would cause too great a change to the LP model too permit the accelerated optimizing implosion technique.) A precise specification of which input data attributes can be changed while preserving an accelerated state is given in Table 6 on page 162, but the following lists should give a general idea:

Some input data attributes that can be changed while preserving an accelerated state:

- Supply volumes.

- Demand volumes.
- Costs, revenues and rewards.
- Weights on objective functions.
- Bound Sets (but with some restrictions; see “Bound Sets and Accelerated Optimizing Implosion” on page 57).

Some input data attributes that cannot be changed while preserving an accelerated state (changing them causes WIT to go into an unaccelerated state):

- Adding a part, demand, operation, BOM entry, substitute BOM entry, or BOP entry.
- Usage rates and yield rates.
- Offsets.

<u>Using Accelerated Optimizing Implosion</u>

This section assumes a familiarity with the use of the API.

To use accelerated optimizing implosion, the application program must do the following: Sometime between calling `witInitialize` and calling `witOptImplode`, the application calls

```
witSetAccAfterOptImp (theWitRun, WitTRUE);
```

which sets the attribute accAfterOptImp to TRUE. This attribute tells WIT to go into an accelerated state at the end of each optimizing implosion. Once this attribute has been set, the first optimizing implosion will not be accelerated, but all subsequent optimizing implosions will be accelerated, provided no input data changes were made that are incompatible with an accelerated state.

By default, the accAfterOptImplode attribute is FALSE. This is because, at the end of an optimizing implosion, WIT occupies considerably more memory if it is in an accelerated state than if it is not. (The accelerated state does not cause WIT to occupy more memory during optimizing implosion; rather, it causes WIT to occupy more memory between optimizing implosions.) Since we did not expect that all applications would be using accelerated optimizing implosion, we decided to set accAfterOptImplode to FALSE by default, which causes WIT to return as much memory as possible to the application after each optimizing implosion.

WIT provides a boolean output attribute, called “accelerated”, that indicates whether or not WIT is in an accelerated state at that moment. The value of the “accelerated” attribute can be obtained by calling the API function `witGetAccelerated`. This attribute may be helpful while you are developing an application, because it enables you to check whether or not WIT is still in an accelerated state after various API functions have been called. Also, for similar reasons, a message is displayed whenever WIT switches between an accelerated and an unaccelerated state.

Bound Sets and Accelerated Optimizing Implosion

This section assumes familiarity with the API and with bound sets.

As indicated above, bound sets are among those input data attributes that can be changed while preserving an accelerated state, but with some restrictions. The behavior of the accelerated state when bound set data is being changed depends on the boolean input attribute, accAfterSoftLB, which can be set by calling the API function, `witSetAccAfterSoftLB`:

- If accAfterSoftLB is TRUE, then all changes to bound sets are compatible with an accelerated state.
- If accAfterSoftLB is FALSE, then all changes to bound sets are compatible with an accelerated state, except those that “soften” a lower bound. A change to a bound set is considered to “soften” a lower bound, if and only if there is some period, t, in which $\text{hardLowerBound}_t = \text{softLowerBound}_t$ before the change and $\text{hardLowerBound}_t < \text{softLowerBound}_t$ after the change. When accAfterSoftLB is FALSE, this kind of change puts WIT into an unaccelerated state. All other changes to bound sets are compatible with an accelerated state. Thus changing hardLowerBound and softLowerBound so that they are equal after the change is compatible with an accelerated state, and changing hardLowerBound or softLowerBound when they were unequal before the change is compatible with an accelerated state, and all changes to hardUpperBound are compatible with an accelerated state.

Clearly, setting accAfterSoftLB to TRUE gives you more flexibility. The disadvantage of setting accAfterSoftLB to TRUE is that it causes WIT to generate a larger LP model, which takes longer to solve (unaccelerated) and occupies more memory. (In tests, we found that setting accAfterSoftLB to TRUE caused WIT to occupy about 30% more memory and take 10-20% more CPU time to solve, but of course, you may get different memory usage and CPU times on your data.)

Setting the value of accAfterSoftLB puts WIT into an unaccelerated state. This is because a different LP model needs to be generated when accAfterSoftLB is TRUE than when it is FALSE. The default value of accAfterSoftLB is FALSE. Thus if you need accAfterSoftLB to be TRUE, you should set it before the first optimizing implosion.

MIP Mode

This feature applies only to optimizing implosion, and it is considered to be an advanced and experimental aspect of WIT.

Normally, optimizing implosion is solved as a Linear Programming (LP) problem, and as a consequence, the various solution attributes (e.g., execution volume) are allowed to take on fractional (non-integers) values. However, in some cases, it may be especially important to require that some of the solution attributes computed by optimzing be required to take on integer values. When this is the case, the “MIP mode” feature may be used, which causes optimizing implosion to be solved as a Mixed Integer Programming (MIP) problem.

To use MIP mode, set the global attribute mipMode to TRUE. (See “mipMode” on page 106.) When this attribute is TRUE, integrality constraints may be specified for the following solution attributes:

- operation.execVol
- subEntry.subVol
- demand.shipVol

Specifically:

- In MIP mode, if the intExecVols attribute for an operation (page 134) is set to TRUE, then optimizing implosion will constrain the execVol of the operation in all periods to take on integer values only.
- In MIP mode, if the intSubVols attribute for a substitute (page 145) is set to TRUE, then optimizing implosion will constrain the subVol of the substitute in all periods to take on integer values only.
- In MIP mode, if the intShipVols attribute for a demand (page 127) is set to TRUE, then optimizing implosion will constrain the shipVol of the demand in all periods to take on integer values only.

NOTES:

1. When mipMode is TRUE, optimizing implosion is solved as a MIP problem; when mipMode is FALSE, optimizing implosion is solved as an LP problem.
2. When mipMode is FALSE, the intExecVols, intSubVols, and intShipVols attributes are ignored
3. MIP mode is an experimental aspect of WIT and should be used with caution: The resulting MIP problem may take vastly longer to solve than the corresponding LP problem without integrality constraints.
4. If optimizing implosion is invoked when mipMode is TRUE, then the following global boolean attributes must be FALSE:
    - accAfterOptImp (page 98)
    - computeCriticalList (page 99)
    - compPrices (page 99)
5. MIP mode applies only to optimizing implosion. It is ignored by heuristic implosion, MRP, FSS, and PIP.
6. If optimizing implosion in MIP mode is invoked on a problem that has no integer variables, WIT will issue a severe error message.

External Optimizing Implosion

This feature applies only to optimizing implosion, it can only be used in API mode, and it is considered to be an advanced and experimental aspect of WIT.

Normally optimizing implosion proceeds as follows:

1. WIT formulates the implosion problem as an LP (or MIP) problem.
2. WIT invokes CPLEX to solve the LP/MIP problem.
3. WIT constructs the implosion solution from the solution to the LP/MIP problem.

In most cases, this is an effective way to solve the optimizing implosion problem. However, in some cases, it may be particularly important to solve the LP/MIP formulation by a different method, either by using solvers different from the one that WIT uses, or by using the same solvers, but in different, perhaps problem-specific ways. In particular, in the MIP case, it may be desireable to add cuts to the formulation or employ branching strategies that are specific to the particular implosion problem.

For cases such as this, the external optimizing implosion feature may be used. In external optimizing implosion, the application program takes responsibilty for solving WIT’s LP/MIP problem. Specifically, external optimizing implosion proceeds as follows:

1. WIT formulates the implosion problem as an LP (or MIP) problem.
2. The application program obtains the LP/MIP problem from WIT.
3. The application program solves the LP/MIP problem, using whatever optimization software is appropriate and whatever specific function calls are appropriate.
4. The application program loads the solution to the LP/MIP problem into WIT.
5. WIT constructs the implosion solution from the solution to the LP/MIP problem.

While ordinary (internal) optimizing implosion is invoked by a calling a single API function, `witOptImplode`, external optimizing implosion involves calling several API functions. Specifically, in the application program, a call to `witOptImplode` would typically be replaced by the following function calls:

- `witStartExtOpt`, which initiates external optimizing implosion. This causes WIT to formulate the implosion problem as an LP or MIP problem.

`witGetExtOptLpProbDbl`, which passes back to the application program a representation of the LP problem, or in MIP mode, the LP relaxation of the MIP problem. The data retrieved by this function is represented in a form suitable to be passed to the COIN OSI function `OsiSolverInterface::loadProblem`.

- In MIP mode, there would be a call to `witGetExtOptIntVarIndices`, which passes back to the application program the set of column indices of the integer variables in the MIP problem. The data retrieved by this function is represented in a form suitable to be passed to the COIN OSI function `OsiSolverInterface::setInteger`.

- Whatever function calls to an LP or MIP solver are required in order to load the problem into the solver, solve the problem, and retrieve the optimal primal solution.
- `witSetExtOptSolnDbl`, which loads the optimal primal solution into WIT. The data passed to this function corresponds to the data retrieved by the COIN OSI function `OsiSolverInterface::getColSolution`.
- `witFinishExtOpt`, which concludes external optimizing implosion. This causes WIT to construct the implosion solution from the solution to the LP/MIP problem.

NOTES:

1. For details on COIN OSI, see http://www.coin-or.org/Doxygen/Osi/class_osi_solver_interface.html.
2. Any LP/MIP solver can be used, as long as its inputs can be constructed from the data provided by WIT's external optimizing implosion functions.
3. When external optimizing implosion is initiated (via `witStartExtOpt`), the following global boolean attributes must be FALSE:
    - accAfterOptImp (page 98)
    - computeCriticalList (page 99)
    - compPrices (page 99)
4. `witGetExtOptLpProb` and `witSetExtOptSoln` are equivalent to `witGetExtOptLpProbDbl` and `witSetExtOptSolnDbl`, but they have arguments of type `float`, while the "`Dbl`" functions have arguments of `double`. Since most solvers have function arguments of type `double`, the "`Dbl`" functions will usually be the ones to use.
5. A call to `witStartExtOpt` puts external optimizing implosion into an "active" state; a call to `witFinishExtOpt`, puts into an "inactive" state. This state can be determined by checking the value of the global boolean attribute "extOptActive", which is TRUE, iff external optimizing implosion is currently active. See "extOptActive" on page 103. The state of external optimizing implosion is important, because many of WIT's API functions may only be called in one of these two states. (Details on this will be given below.)
6. As an alternative to `witFinishExtOpt`, the function `witShutDownExtOpt` can be called. `witShutDownExtOpt` puts external optimizing implosion into an inactive state, but without constructing the implosion solution from the solution to the LP/MIP problem. This may be useful if the application program was not able to provide a solution to the LP/MIP problem.

Sometimes, when working in external optimizing implosion mode, it may be desireable to make calls to the solver specifying individual rows or columns in the constraint matrix. For example, in MIP mode, the application program might need to specify columns in a branching strategy. To do this intelligently, it is necessary for the application program to have access to WIT's mapping from the objects of the implosion problem to the rows and columns of the matrix. WIT provides a set of API functions that specify this mapping. For example, the function `witGetOperationExecVarIndex` passes back to the application program the column index of the execution variable for a particular operation in a particular period, while the function `witGetPartResourceConIndex` passes back to the application program the row index of the resource allocation constraint for a particular part in a particular period. For a precise specification of WIT's LP/MIP formulation, see [2].

WIT's column index functions are the following:

- `witGetPartScrapVarIndex`
- `witGetPartStockVarIndex`
- `witGetDemandShipVarIndex`
- `witGetDemandCumShipVarIndex`
- `witGetOperationExecVarIndex`
- `witGetBomEntryNonSubVarIndex`
- `witGetSubsBomEntrySubVarIndex`
- `witGetPartStockSlbvVarIndex`
- `witGetDemandCumShipSlbvVarIndex`
- `witGetOperationExecSlbvVarIndex`

WIT's row index functions are the following:

- `witGetPartResourceConIndex`
- `witGetDemandShipConIndex`
- `witGetBomEntrySubConIndex`
- `witGetPartStockSlbConIndex`
- `witGetDemandCumShipSlbConIndex`
- `witGetOperationExecSlbConIndex`

For more information about the column and row index functions, see "witGet*VarIndex" on page 406 and "witGet*ConIndex" on page 410.

The following functions may be called if and only if external optimizing implosion is currently active:

- `witSetExtOptSoln`
- `witSetExtOptSolnDbl`
- `witFinishExtOpt`
- `witShutDownExtOpt`

- `witGetExtOptLpProb`
- `witGetExtOptLpProbDbl`
- `witGetExtOptIntVarIndices`
- `witGet*VarIndex`
- `witGet*ConIndex`

where `*` represents any string of characters. Calling any of these functions when external optimizing implosion is inactive will result in a severe error.

When external optimizing implosion is active, the following are the only functions that may be called:

- `witSetExtOptSoln`
- `witSetExtOptSolnDbl`
- `witFinishExtOpt`
- `witShutDownExtOpt`
- `witGet*`
- `witInitialize`
- `witDeleteRun`
- `witResetObjItr`
- `witAdvanceObjItr`
- `witSetMesg*`

where `*` represents any string of characters. Calling any other function when external optimizing implosion is active will result in a severe error.

Disallowing Scrap

In some applications of optimizing implosion, it may be necessary to disallow scrapping of some of the parts in the implosion problem. To disallow scrapping of a specific part in optimizing implosion, set the scrapAllowed attribute to FALSE for the part. (See “scrapAllowed” on page 123.) The scrapAllowed attribute is a boolean attribute for parts. Its default value is TRUE. When it is set to FALSE for a particular part, the scrapVol of the part will be constrained to be zero in optimizing implosion.

Note that the scrapAllowed attribute is ignored by heuristic implosion. (The heuristic cannot avoid generating a positive scrapVol if it needs to.)

Multiple Objectives Mode

By default, optimizing implosion works with a single objective function: It finds an implosion solution that achieves the maximum feasible value of a single linear function of the decision variables that make up the solution. In some cases, it may be desirable to have optimizing implosion find an implosion solution that achieves the maximum feasible value of several linear objective functions, where the objective functions are treated as a strict hierarchy. You can do this in WIT by using “multiple objectives mode”.

When WIT is in multiple objective mode, one or more objectives are defined and each is given a name to identify it. Each attribute that specifies an aspect of the objective is allowed to have a different value for each objective. Thus for example, if there are 3 objectives, the execCost for any given operation in any period will take on 3 distinct values.

The objectives are given a rank ordering. That is, there’s a first objective, a second objective, etc. When optimizing implosion is invoked, an implosion solution is found that achieves the “lexicographic maximum value” of the objective functions with respect to this rank ordering. Specifically, this means the following:

1. The implosion solution will achieve the maximum feasible value of the first objective.
2. The implosion solution will achieve the maximum feasible value of the second objective, subject to condition 1.
3. The implosion solution will achieve the maximum feasible value of the third objective, subject to conditions 1 and 2.
4. etc.

To put WIT into multiple objectives mode, set the global boolean attribute “multiObjMode” to TRUE. (See “multiObjMode” on page 107.) When this attribute is TRUE, WIT is in multiple objectives mode; when it is FALSE, WIT is in single objective mode (the default).

Once WIT has been put into multiple objectives mode, the set of objectives should be specified. This can be done by setting the global attribute “objectiveList”. (See “objectiveList” on page 109.) The value of this attribute is a list of strings, one for each objective, where each string is taken to be the name of the corresponding objective. The objective names must be distinct from each other and must not contain the character “|” (vertical bar).

Alternatively, the set of objectives can be specified by setting the global attribute objectiveListSpec”. (See “objectiveListSpec” on page 109.) The value of this attribute is a single string that is taken to be the concatenation of all the objective names, with each separated by a “|” (vertical bar) character. Thus “Reward|ExecCost|OtherCost” specifies three objectives: “Reward”, “ExecCost”, and “OtherCost”. Whenever either of the objectiveList or objectiveListSpec attributes is set, WIT automatically sets the other one to match it.

In multiple objective mode, each of the following attributes is objective-specific, that is, it has (potentially) distinct values for each objective:

- scrapCost for parts

- stockCost for parts
- execCost for operations
- subCost for substitutes
- shipReward for demands
- cumShipReward for demands
- objValue (global)
- objectiveSeqNo (global) (see below)

At any point in time, one of the objectives is considered to be the “current objective”. You specify the current objective by setting the value of the global string attribute `currentObjective”. (See “currentObjective” on page 101.) The value of this attribute is the name of the current objective.

When you set the value of an objective-specific attribute, WIT only sets the value for the current objective; the value for all other objectives is left unchanged. Similarly, when you retrieve the value of an objective-specific attribute, WIT retrieves the value for the current objective. For example, if you set the scrapCost of some part to some vector of values, WIT sets the scrapCost of the part to that vector for the current objective only; the scrapCost of the part for all other objectives is left unchanged. Thus the input data that defines each objective can be specified by setting the current objective and then setting the cost and reward attributes for the current objective. This approach can be used in an application program or in an input data file.

The rank ordering of the objectives is specified by the global attribute “objectiveSeqNo”. (See “objectiveSeqNo” on page 109.) The value of this attribute is required to be an integer from 1 to the number of objectives. This attribute is objective-specific, so you set its value by first setting the current objective to the objective in question. When optimizing implosion is invoked, the objectiveSeqNo of each objective must be distinct from all the other objectives. (But note that this restriction does not apply until optimizing implosion is actually invoked.) When optimizing implosion is invoked, it will find an implosion solution that achieves the lexicographic maximum value of the objective functions with respect to the rank ordering defined by the objectiveSeqNo value of each objective. Thus

1. The objective whose objectiveSeqNo is 1 will be at its maximum feasible value.
2. The objective whose objectiveSeqNo is 2 will be at its maximum feasible value, subject to condition 1.
3. The objective whose objectiveSeqNo is 3 will be at its maximum feasible value, subject to conditions 1 and 2.
4. etc.

By default, the objectives are ranked in the order in which they appear in the objectiveList.

Soft lower bounds are treated differently in multiples objectives mode than in single objective mode. Recall that in single objective mode, one aspect of the objective function is:

- wbounds ∗ boundsValue

where boundsValue is the value of the bounds objective, i.e., the sum of the violations of all soft lower bounds.

In multiple objectives mode, the user-specified objectives do not include the above term. In fact, the wbounds attribute is ignored altogether. Instead, if the data contains at least one non-trivial soft lower bound (i.e., one that is strictly greater than the corresponding hard lower bound), the lexicographic optimization is extended to minimize the bounds objective, with a rank order that precedes all of the user-specified objectives. Thus in the presence of non-trivial soft lower bounds, optimizing implosion will find an implosion solution with the following characteristics:

1. The bounds objective will be at its minimum feasible value.
2. The objective whose objectiveSeqNo is 1 will be at its maximum feasible value, subject to condition 1.
3. The objective whose objectiveSeqNo is 2 will be at its maximum feasible value, subject to conditions 1 and 2.
4. etc.

NOTES

1. In tests, the CPU time for optimizing implosion in multiple objectives mode tended to grow roughly linearly with the number of objectives.
2. The lexicographic optimization uses a numerical optimality tolerance given by the global attribute multiObjTol. (See “multiObjTol” on page 107.)
3. The following attributes may only be set when no parts or operations have been created:
   - multiObjMode
   - objectiveList
   - objectiveListSpec
4. The following global attributes may only be set or retrieved in multiObjMode:
   - objectiveList
   - objectiveListSpec
   - currentObjective
   - objectiveSeqNo
   - multiObjTol
5. The objectiveList attribute cannot be set from an input data file; in that context, the objectiveListSpec attribute should be used instead.

6. The following actions are not allowed in multiple objectives mode:

- Setting the global attribute stochMode to TRUE.
- Calling `witStartExtOpt`.
- Calling `witCopy...Data`, e.g., `witCopyPartData`. (However `witCopyData` is allowed.)
- Invoking optimizing implosion when any of the following global attributes is TRUE: computeCriticalList, compPrices, or accAfterOptImp.

CPLEX Parameter Specifications

NOTE: This feature requires explicit knowledge of CPLEX and access to its user documentation in order to be used properly.

When CPLEX is being invoked directly (i.e., not through WIT), one important aspect of using it is the use of its “parameters”. The parameters of CPLEX are the means by which the behavior of CPLEX can be customized. Conceptually, the parameters of CPLEX are similar to the global attributes of WIT: Each parameter has a name, a data type (int, long, double, or string), a value matching the data type and a default value, and its value can be set or retrieved at any time by the CPLEX application program. For example, there is a CPLEX parameter whose name is CPX_PARAM_LPMETHOD and has type int and a default value of 0. This parameter specifies which algorithm is to be used to solve LP problems. (For example, 1 indicates primal simplex.) For complete information on CPLEX parameters, consult the CPLEX documentation, the section entitled: “Parameters of CPLEX”.

WIT provides a mechanism for setting the parameters in CPLEX before CPLEX is invoked. The approach is somewhat indirect, and this is due to the fact that at the point in time at which a WIT API function is being called, WIT has not yet set up a CPLEX environment and consequently cannot actually communicate with CPLEX. Instead, WIT stores a set of “CPLEX parameter specifications”. Each CPLEX parameter specification has a name, a data type, and a value matching the data type. The data type is either int or double. CPLEX parameters of type long are handled by WIT as if they were of type int. WIT does not provide access to string CPLEX parameters. Initially, there are no CPLEX parameter specifications stored. At any time prior to calling optimizing or stochastic implosion, the user / application program can create one or more CPLEX parameter specifications in WIT. Then, during optimizing or stochastic implosion, just before it calls one of the CPLEX solve routines (either `CPXlpopt` or `CPXmipopt`), WIT goes through each CPLEX parameter specification that was created and sets the corresponding CPLEX parameter to the value given in the specification.

There are two ways to work with WIT’s CPLEX parameter specifications: explicitly through the API, or implicitly through attributes. To begin, consider the explicit API approach. To create a CPLEX parameter specification of type int, call the API function `witAddIntCplexParSpec`. This function takes three arguments: a WitRun (see chapter 4), a string, and an int. It creates a new CPLEX parameter specification of type int with the string as its name and the int as its value.

Consider the following WIT application code:

```
witAddIntCplexParSpec (
   theWitRun,
   "LPMETHOD",
   1);
```

This code will cause WIT to create a CPLEX parameter specification of type int with the name “LPMETHOD” and value 1. As a result, during optimizing or stochastic implosion with CPLEX, WIT will call CPLEX to set the value of the integer CPLEX parameter CPX_PARAM_LPMETHOD to 1, which will cause CPLEX to use the primal simplex method to solve any LP problems.

Similarly, to create a CPLEX parameter specification of type double, call the API function `witAddDblCplexParSpec`. This function takes three arguments: a WitRun, a string, and a float. It creates a new CPLEX parameter specification of type double with the string as its name and the float as its value. (The float is converted to a double.) There is also a double precision variant of this function, `witAddDblCplexParSpecDbl`, which takes a double as its third argument and has the same behavior. (See “Double Precision Functions” on page 389.)

NOTES:

- Whenever a new CPLEX parameter specification a created, any existing CPLEX parameter specification with the same name (of either type) is deleted.
- The name of a CPLEX parameter specification should (when prefixed with “CPX_PARAM_”) match the name of a CPLEX parameter as defined for the CPLEX callable library.
- If the name of a CPLEX parameter specification does not match the name of any CPLEX parameter, this will be detected during optimizing or stochastic implosion with CPLEX, and WIT will issue a severe error message at that time.
- If the name of a CPLEX parameter specification is given as “NO_PARAM”, WIT will issue a severe error message immediately.
- If the type of a CPLEX parameter specification does not match the type of the corresponding CPLEX parameter, this will be detected during optimizing or stochastic implosion with CPLEX, and WIT will issue a severe error message at that time.
- To set a CPLEX parameter of type long, create a CPLEX parameter specification of type int.

- To retrieve the value of a CPLEX parameter specification of type int, call `witGetIntCplexParSpec`. To retrieve the value of a CPLEX parameter specification of type double, call `witGetDblCplexParSpec` or `witGetDblCplexParSpecDbl`.
- To clear (delete) all existing CPLEX parameter specifications in WIT, call `witClearCplexParSpecs`.
- WIT sets the value of the CPX_PARAM_LPMETHOD parameter itself, using its own internal logic. However, if there is a CPLEX parameter specification with the name "LPMETHOD", it will override WIT's setting of this parameter.
- Some CPLEX parameters can cause the CPLEX solve routines to terminate early, with no feasible solution. In this case, WIT will issue a severe error message that displays the text that it received from CPLEX indicating the solution status. e.g., "time limit exceeded, no integer solution".
- Some CPLEX parameters can cause the CPLEX solve routines to terminate early, with a solution that's feasible, but not optimal. In this case, if WIT does not need an optimal solution, it will issue a warning message that displays the text that it received from CPLEX indicating the solution status. e.g., "time limit exceeded". After issuing the warning, it will continue as normal, using the feasible solution from CPLEX to compute an implosion solution. If WIT does need an optimal solution (for example, to compute the critical part list from the dual solution), it will issue a severe error message that displays the text that it received from CPLEX indicating the solution status. e.g., "time limit exceeded".
- Since some CPLEX parameters can cause the CPLEX solve routines to terminate early, it may be helpful for the application program to check the CPLEX solution status code from the solve. This value is stored in the WIT attribute cplexStatusCode (page 101). Also relevent in this case are the attributes cplexStatusText (page 101), cplexMipBound (page 99) and cplexMipRelGap (page 99).

An alternative way to work with WIT's CPLEX parameter specifications is through WIT data attributes. There are three global input attributes that pertain to CPLEX parameter specifications:

- cplexParSpecName: type string, default value "NO_PARAM" (see page 100)
- cplexParSpecIntVal: type int (see page 100)
- cplexParSpecDblVal: type float (see page 100)

These are ordinary attributes: Their values can be set and retrieved through the API and can be set from a WIT data file. However, setting the value of cplexParSpecIntVal has the following side effect: If the current value of cplexParSpecName is anything other than "NO_PARAM", a new CPLEX parameter specification of type int will be immediately created, whose name will be the value of cplexParSpecName and whose value will be the specified value of cplexParSpecIntVal. Similarly, setting the value of cplexParSpecDblVal has the following side effect: If the current value of cplexParSpecName is anything other than “NO_PARAM”, a new CPLEX parameter specification of type double will be immediately created, whose name will be the value of cplexParSpecName and whose value will be the specified value of cplexParSpecDblVal.

This way of working with CPLEX parameter specifications is less generally convenient than the explicit API approach, but its advantage is that it can be used from a WIT data file. For example, you can instruct WIT to set the value of the CPLEX parameter CPX_PARAM_LPMETHOD to 1 during optimizing or stochastic implosion with CPLEX by putting the following lines into a WIT data file:

```
set problem cplexParSpecName   "LPMETHOD";
set problem cplexParSpecIntVal 1;
```

Automatic Priority

This feature is available in heuristic mode only. When using this feature WIT will generate priorities used in heuristic implosion from the objective function data.

To have heuristic implosion use automatic priority, set the autoPriority attribute to TRUE. See “autoPriority” on page 98.

Equitable Allocation Heuristic Implosion

If the “equitability” attribute is larger than 1, heuristic implosion performs an equitable allocation. The Equitable Allocation Heuristic was developed to provide an equitable allocation of materials and capacity among a set of equal priority demands.

The Equitable Allocation Heuristic is identical to the priority allocation heuristic in all respects except one. When the priority allocation heuristic must decide between two or more demand volumes with the same demand period being considered for shipment in the same shipment period and the priorities are the same, the priority allocation heuristic breaks the tie by giving preference to the demand that was entered earlier into WIT. In this case, the equitable allocation heuristic makes N cycles through each set of “tied” demands, attempting to allocate approximately (100/N)% of each the demand on each cycle, where N is the user specified “equitability” parameter.

Consider the following example:

Suppose you have three demands for the same part, where the demand volumes in a given period are:

- d1 = 200

- d2 = 200
- d3 = 100

Furthermore, assume that these demands have equal priority and that they were entered in the order given. Finally, suppose that there is only enough supply to ship 400 units of the product. The priority allocation heuristic (non-equitable) would ship:

- s1 = 200
- s2 = 200
- s3 = 0

However, using equitable allocation the shipments would be more “equitable”:

- s1 = 160
- s2 = 160
- s3 = 80

NOTES:

1. NOTE: In this simple and trivial example, the allocation is “purely” equitable, (i.e., an equal proportion, 80%, of each demand was met.) In more complex situations, the allocations will be “approximately” equitable. The higher the value of the data attribute, `equitability`, the more equitable the allocation will be, at the expense of increased run time.
2. Consider the situation in which a demand with a unique priority is encountered, i.e., there is no tie for priority between the demand and any other demand in the same period. Normally, WIT will allocate this demand in a single pass, avoiding the multi-pass logic of equitable allocation. (This reflects the fact that equitable allocation is not an issue in this case.) This logic can be overridden by the user; see “forcedMultiEq” on page 103.

Heuristic Allocation

This feature is available in API mode only and the discussion assumes familiarity with the API.

The purpose of the heuristic allocation feature is to enable an application developer to build his/her own customized heuristic implosion algorithm by invoking the same functions that are used by WIT’s heuristic implosion. There are six API functions and one attribute specifically associated with heuristic allocation. The functions are:

```
witStartHeurAlloc
witIncHeurAlloc
witEqHeurAlloc
witFinishHeurAlloc
witShutDownHeurAlloc
witGetHeurAllocActive
```

The attribute is:

heurAllocActive

An application program can use these functions to implement a customized heuristic implosion in either of two ways: with and without equitable allocation. (The concept of equitable allocation is explained in “Equitable Allocation Heuristic Implosion” on page 69.) Heuristic allocation without equitable allocation is somewhat easier to use and will be described first.

<u>Heuristic Allocation without Equitable Allocation</u>

To implement a customized heuristic implosion without equitable allocation, an application program would use the heuristic allocation API functions in the following way:

1. Invoke `witStartHeurAlloc` once. This function initiates heuristic allocation. Specifically, it
   - Invokes preprocessing, if necessary.
   - Sets the execution and shipment schedules to zero.
   - Sets the critical part list to the empty list.
   - Initializes the internal data structures for heuristic allocation.
   - Puts heuristic allocation into an “active” state.
2. Invoke `witIncHeurAlloc` many times. This function “increments” heuristic allocation. The function has several arguments. Conceptually, you pass it a demand, a shipment period (shipPeriod), and a desired incremental shipment volume (desIncVol). The function then (heuristically) attempts to increase the shipVol of the specified demand in shipPeriod by as much as possible, up to desIncVol, subject to keeping the execution and shipment schedules feasible. To do this, it will alter the execution schedule as needed, but keep the shipment schedule fixed, except for the shipVol of the specified demand in shipPeriod. Finally, `witIncHeurAlloc` passes back an argument (incVol), which is the amount by which the shipVol of the specified demand in shipPeriod was increased.
3. Invoke `witFinishHeurAlloc` once. This function concludes heuristic allocation. Specifically, it
   - Puts heuristic allocation into an “inactive” state.
   - Releases the memory resources used by heuristic allocation.
   - Invokes postprocessing.

Once `witFinishHeurAlloc` has been invoked, WIT is in a state similar to the conclusion of heuristic implosion: The implosion solution is available, the critical parts list is available (if computeCriticalList is TRUE), the results of postprocessing are available, and FSS can be invoked.

At any point in time, heuristic allocation is in one of two possible states: “active” or “inactive”. The active state indicates that the internal data structures for heuristic allocation are currently set up. This state is a requirement in order for `witIncHeurAlloc` to function correctly and so WIT issues a severe error if `witIncHeurAlloc` is invoked when heuristic allocation is in an inactive state. Invoking `witStartHeurAlloc` puts heuristic allocation into an active state. Any of the following actions will put heuristic allocation into an inactive state:

- Invoking `witFinishHeurAlloc.`
- Invoking `witShutDownHeurAlloc.` (See below.)
- Setting the value of any input data.
  (See also “Modifiable Heuristic Allocation” on page 74.)
- Invoking postprocessing.

The state of heuristic allocation is indicated by the boolean attribute heurAllocActive: Its value is TRUE if and only if heuristic allocation is in an active state. Its value can be queried by invoking `witGetHeurAllocActive`. Also, a message is displayed whenever its value is changed.

The key to building a customized heuristic implosion is, of course, in choosing the exact sequence of `witIncHeurAlloc` calls. This determines which demands and periods take precedence over others, how backlogging is handled/prioritized, etc.

As an alternative to `witFinishHeurAlloc`, one can invoke `witShutDownHeurAlloc`, which has a similar effect:

- It puts heuristic allocation into an “inactive” state.
- It releases the memory resources used by heuristic allocation.

The differences between `witFinishHeurAlloc` and `witShutDownHeurAlloc` are as follows:

- `witFinishHeurAlloc` invokes postprocessing, while `witShutDownHeurAlloc` does not.
- If respectStockSLBs is TRUE (page 113), `witFinishHeurAlloc` will attempt to satisfy stock soft lower bounds, while `witShutDownHeurAlloc` will not.

`witShutDownHeurAlloc` may particularly helpful when using the “User-Specified Heuristic Starting Solution” capability. (page 75)

Equitable Heuristic Allocation

The procedure to implement a customized heuristic implosion with equitable allocation is the same as the procedure without equitable allocation, except that you use `witEqHeurAlloc` in place of `witIncHeurAlloc`.

To see how to use `witEqHeurAlloc`, define an “allocation target” (conceptually) to be a data object consisting of a demand, a shipment period (shipPeriod), and a desired incremental shipment volume (desIncVol), i.e., the arguments to `witIncHeurAlloc`. The main argument to `witEqHeurAlloc` is an ordered list of allocation targets. The function (heuristically) attempts to increase the shipVols of the specified demands in the specified shipPeriods by as much as possible, up to the desIncVols, subject to keeping the execution and shipment schedules feasible. To do this, it will alter the execution schedule as needed, but keep the shipment schedule fixed, except for the shipVols of the specified allocation targets. Finally, `witEqHeurAlloc` passes back a list specifying the amount by which the shipVols of the specified allocation targets were increased.

The difference between calling `witEqHeurAlloc` for a group of allocation targets and calling `witIncHeurAlloc` once for each target occurs if there are resource conflicts between the targets. In the `witIncHeurAlloc` case, such conflicts are resolved in favor of the first target for which `witIncHeurAlloc` was called. In the `witEqHeurAlloc` case, an attempt is made to allocate in proportion to the desIncVols. This is done by making N passes through the specified targets, where N is the value of the “equitability” attribute. At each pass, approximately (100/N)% of the desIncVol is allocated. Thus higher values of equitability would tend to result in better approximations to equitable allocation, at the cost of longer computational run times.

Similarities Between Heuristic Allocation and Heuristic Implosion

WIT’s heuristic implosion is implemented by using (equitable) heuristic allocation in the manner described above (using internal versions of the API functions). Consequently, heuristic allocation shares many of the properties of heuristic implosion:

1. The resulting implosion solution will be feasible.
2. Heuristic allocation will never plan to ship early.
3. Heuristic allocation uses lot sizing.
4. Heuristic allocation won't build if it has sufficient stock.
5. Heuristic allocation uses build-ahead in the same manner as with heuristic implosion.
6. Heuristic allocation uses available supply of substitutes, but does not build parts to use as substitutes. Heuristic allocation considers the substitutes for a BOM entry in the order in which they were input to WIT.

Comments About Heuristic Allocation

1. Heuristic allocation ignores priority. Priority logic is a responsibility of the application program.
2. Since the shipment period is always explicitly specified by the application program, heuristic allocation ignores shipLateUB.

3. When selecting the value of desIncVol, remember that it may it be allowable for the shipVol of a demand in a period to be larger than the demandVol in that period, provided there is some backlog. The only feasibility constraint that applies here is that the cumulative shipment to a demand in a period must be ≤ the cumulative demand in the period. And even so, if desIncVol is set large enough to violate this constraint, WIT does not issue an error, it simply sets incVol small enough to satisfy this constraint.
4. If the list of allocation targets passed to `witEqHeurAlloc` contains no more than one positive desIncVol, it will normally make only one pass through the allocation targets, allocating the entire desIncVol at once. (This reflects the fact that equitable allocation is not an issue in this case.) This logic can be overridden by the user; see “forcedMultiEq” on page 103.
5. Note that the order in which the allocation targets appear in the list passed to `witEqHeurAlloc` can be significant: If there is any deviation from a pure equitable allocation, the “inequity” is in favor of allocation targets that appear earlier in the list.
6. It is permissible to intermix calls to `witIncHeurAlloc` with calls to `witEqHeurAlloc`.

Modifiable Heuristic Allocation

Normally, the application program is not allowed to alter or modify the implosion solution while heuristic allocation is active. For example, if `witSetOperationExecVol` is invoked during heuristic allocation, WIT will immediately deactivate heuristic allocation and then set the execVol attribute. However, for some sophisticated applications, it may be desirable to override this default behavior. The “modifiable heuristic allocation” capability allows this to be done.

To use the modifiable heuristic allocation capability, set the global boolean attribute modHeurAlloc to TRUE. (See “modHeurAlloc” on page 106.) This must be done before calling `witStartHeurAlloc`. Calling `witStartHeurAlloc` when modHeurAlloc is TRUE activates modifiable heuristic allocation. Modifiable heuristic allocation is the same as ordinary heuristic allocation, except that the values of the following 3 attributes can be set without deactivating it:

- execVol for operations
- subVol for substitutes
- shipVol for demands

The implosion solution that results from setting these attributes is required to be feasible. See “Testing the Feasibility of a User-Specified Solution” on page 38 for WIT’s feasibility criteria. WIT does not verify the feasibility of the implosion solution every time it is modified. Instead, it waits until an API function is called that causes the heuristic allocation algorithm to continue.

Specifically, the following API functions cause WIT to do a feasibility check during modifiable heuristic allocation:

- `witIncHeurAlloc`
- `witIncHeurAllocTwme`
- `witEqHeurAlloc`
- `witEqHeurAllocTwme`
- `witFinishHeurAlloc`
- `witShutDownHeurAllocTwme`
- Any function that causes heuristic allocation to stop being active.

Thus multiple calls may be made to `witSetOperationExecVol`, etc. before feasibility is verified. This allows the application program to modify the implosion solution in a complex way, without needing to maintain feasibility with each API function call.

The feasibility checking is only performed on individual constraints that have the potential to be violated by the specific modifications that were made to the implosion solution. Thus if only a few modifications are made, then only a few feasibility constraints are checked. This prevents the CPU time for heuristic allocation from drastically increasing when the implosion solution is modified.

Finally, note that when `witStartHeurAlloc` is called while modHeurAlloc is TRUE, the following global boolean attributes are required to have the values given here:

- selectionRecovery must be TRUE.
- skipFailures must be FALSE.
- perfPegging must be FALSE.
- computeCriticalList must be FALSE.
- pgdCritListMode must be FALSE.

User-Specified Heuristic Starting Solution

This feature allows the user or application program to specify a starting solution for heuristic implosion/allocation. To use this feature, set the global boolean attribute userHeurStart to TRUE. (See “userHeurStart” on page 117.) When userHeurStart is TRUE and heuristic implosion or heuristic allocation is invoked, the heuristic uses the user-specified solution as its initial solution. (See “User-Specified Solution” on page 37.) It then proceeds from there, subject to the following constraint: For each operation, substitute and demand, the execVol, subVol and shipVol that it computes will each be ≥ the corresponding execVol, subVol and shipVol in the user-specified solution.

When heuristic implosion/allocation is invoked in userHeurStart mode, the user-specified solution needs to be feasible. If it is not, a warning message is issued and the solution computed by the heuristic is likely to be infeasible in the same way. See “Testing the Feasibility of a User-Specified Solution” on page 38 for WIT’s feasibility criteria.

Concurrent Pegging

“Concurrent pegging” is a pegging technique that can be used as an alternative to post-implosion pegging. (See “Post-Implosion Pegging” on page 43 for a definition of pegging.) Unlike post-implosion pegging, concurrent pegging can only be used with heuristic implosion/allocation. The concurrent pegging technique forms its pegging by monitoring the process by which the heuristic computes the solution. Specifically, each time an operation’s execVol is being increased, this is being done in order to enable the shipVol of some demand to be increased in some period. The concurrent pegging feature internally records the association between the increase in the execVol and the demand and period whose shipVol is being increased. It also records the association between any increase in a substitute’s subVol and the corresponding demand and period whose shipVol is being increased. These associations can then be retrieved using API functions.

To have heuristic implosion and allocation perform concurrent pegging, set the global perfPegging attribute to TRUE. See “perfPegging” on page 111.

To retrieve the execVol pegging, call `witGetDemandExecVolPegging`. This function works similarly to `witGetDemandExecVolPip`. When calling this function, the application program specifies a demand and shipment period. The function returns several lists that, in effect, constitute a list of “pegging triples”. Each pegging triple (for execVols) specifies an operation, an execution period and a “pegged execVol”. The pegged execVol is the total amount by which the execVol of the operation was increased in the execution period in order to increase the shipVol of the demand in the shipment period, since the last time the pegging was cleared. (Clearing the pegging will be explained below.) The set of pegging triples returned by `witGetDemandExecVolPegging` corresponds to the set of all operations and execution periods whose execVols were increased in order to increase the shipVol of the demand in the shipment period, since the last time the pegging was cleared.

To retrieve the subVol pegging, call `witGetDemandSubVolPegging`, which works similarly to `witGetDemandExecVolPegging`. Unlike post-implosion pegging, concurrent pegging does not provide pegging for supplyVol, consVol, or prodVol, and there is no concept of sideVol in concurrent pegging.

Various actions have the effect of “clearing” the pegging, i.e., deleting all currently existing pegging triples (for both execVols and subVols). The following functions clear the pegging:

- `witClearPegging`
- `witSetPerfPegging` (when setting it to FALSE)
- `witHeurImplode` (during its initialization)

- `witStartHeurAlloc`
- `witPurgeData`
- `witCopyData` (clears the pegging of the destination WitRun)
- `witInitialize`
- `witDeleteRun`

NOTES:

1. The functions `witGetDemandExecVolPegging` and `witGetDemandSubVolPegging` are normally called after heuristic implosion and heuristic allocation or during heuristic allocation (i.e., between calls to `witIncHeurAlloc` or `witEqHeurAlloc`).
2. If the shipVol for a demand in a period has not increased since the last time the pegging was cleared, there will be no pegging triples associated with that demand in that period.
3. The complete pegging can potentially require a large amount of memory. If this is a problem, the memory requirement can be kept to minimum by calling `witClearPegging` to clear the pegging (after retrieving the desired pegging information) frequently during heuristic allocation.
4. In some cases, it may be logically useful to clear the pegging during heuristic allocation. For example, the following sequence of function calls returns the pegging for a single call to `witIncHeurAlloc`:

```
witClearPegging (...);
witIncHeurAlloc (...);
witGetDemandExecVolPegging (...);
witGetDemandSubVolPegging (...);
```

5. Unlike PIP. the pegging produced by concurent pegging adheres to the lot size constraints, since the heuristic does so.
6. Unlike PIP, current pegging does not generally have the feasible partitioning property (even with respect to an auxiliary problem). In some cases, the part(s) produced by the operation for a pegged execVol might not be used (in any direct sense) by the demand to which the execVol is pegged. Instead, the heuristic might be executing the operation in order to enable it to reallocate to the demand resources that had previously been allocated to a different demand. For an example of this, see “Stock Reallocation” on page 83. In all cases, an execVol is pegged to the demand whose shipVol was increased as a result of increasing the execVol, and not necessarily the demand that actually uses the output of the operation. (The same comment applies to pegged subVols.)

Pegged Criticial List

This capability is available only in heuristic mode. If requested, the heuristic will generate a “pegged critical list”. This is similar to the critical parts list (page 38), but provides additional information.

The pegged critical list is an ordered list in which each element consists of the following four components:

- A critical part
- A critical period
- A demand
- A shipment period

The presence of an element in the pegged critical list indicates the following: At some point during the execution of the heuristic, a shortage in the supplyVol of the critical part in the critical period prevented (at least temporarily) a potential increase to the shipVol of the demand in the shipment period. The elements of the pegged critical list appear in the order in which the shortages were encountered during the execution of the heuristic.

To have the heuristic generate the pegged critical list, set the pgdCritListMode attribute to TRUE. (See “pgdCritListMode” on page 112.) To retrieve the pegged critical list, invoke `witGetPgdCritList`.

NOTES

1. The pegged critical list cannot be generated if either of the following conditions hold:
    - singleSource = TRUE for any part or BOM entry
    - selSplit = TRUE

Multiple Routes

This feature only applies to heuristic implosion and allocation.

The complete BOM structure used by WIT includes two kinds of “multiple choice structures”:

- If there is more than one BOP entry that produces a part, one can use any or all of the BOP entries to produce the part.
- If a BOM entry has one or more substitutes associated with it, one can use any or all of the substitutes in place of, or in addition to, the BOM entry itself.

When heuristic implosion encounters a multiple choice structure, its default behavior is to make a single selection. This is called the “single route” technique, because it only takes one route at each multiple choice structure:

- If there is more than one BOP entry that produces a part, the single route technique will use only one of the BOP entries.
- If a BOM entry has one or more substitutes associated with it, the single route technique will use the substitutes as well as the BOM entry, but it will only use available supply of the substitute parts. It will not build the consumed parts to be used as substitutes.

In contrast to the single route technique, the “multiple routes” technique takes as many routes as necessary at each multiple choice structure:

- If there is more than one BOP entry that produces a part, the multiple routes technique will use as many of the BOP entries as necessary.
- If a BOM entry has one or more substitutes associated with it, the multiple routes technique will use as many of the substitutes as necessary, building the consumed parts as needed.

To have heuristic implosion and allocation use the multiple routes technique, set the multiRoute attribute to TRUE. See “multiRoute” on page 107.

<u>Comments About the Multiple Routes Technique</u>

1. The multiple routes technique considers the routing alternatives sequentially: For any given part, it starts by using one of the BOP entries that produces the part and then switches to a different BOP entry and so on. Similarly, for any given BOM entry, the multple routes technique starts by using the BOM entry itself and then switches to one of the BOM entry’s substitutes and so on.
2. The multiple routes technique is a heuristic: It may be possible to use the multiple choice structures in ways that are more effective at meeting demands than the selections made by this heuristic.
3. A disadvantage of using the multiple routes technique is that it is slower and uses more memory than the single route technique. How much more time and memory it is likely to use is not known at this time.
4. The following input attributes allow the user to control the multiple routes technique:
    - multiRoute (Global)
    - expAllowed (BOP entries)
    - expAllowed (Substitute BOM entries)
    - expAversion (BOP entries)
    - expNetAversion (Substitute BOM entries)

    For information on data attributes, see chapter 2.

Proportionate Routing

This feature only applies to heuristic implosion and allocation.

The proportionate routing technique applies to the same “multiple choice structures” as the multiple routes technique:

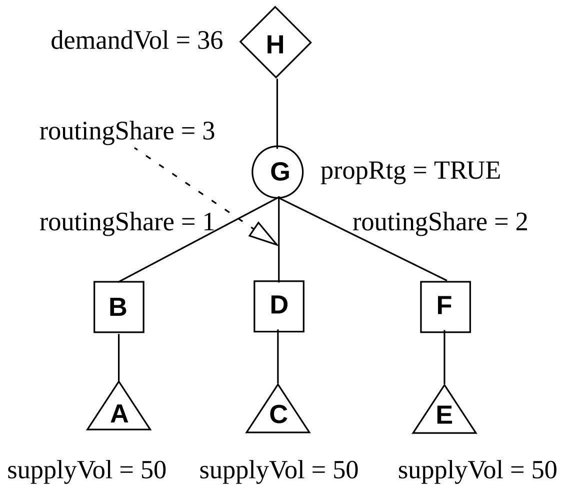
- A part produced by more than one BOP entry
- A BOM entry with at least one substitute

For clarity, the “part” case will be discussed first.

To request proportionate routing for a particular part in a particular period, set the “propRtg” attribute on the part to TRUE in the period. (See “propRtg” on page 122.) As previously indicated. the multiple routes technique uses each BOP entry for a part in sequence. In contrast, the proportionate routing technique uses all of the BOP entries for the part at the same time, initially, according to fixed ratios specified by the user. The ratios are determined by BOP entry attribute, “routingShare”. (See “routingShare” on page 151.)

Consider the following example:

FIGURE 12 First Example of Proportionate Routing



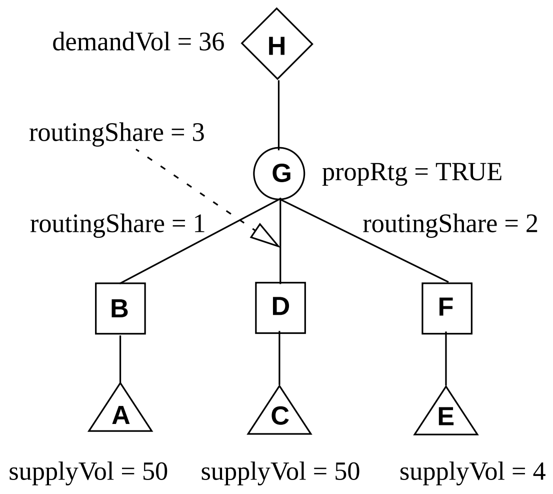
In this case, the heuristic will meet the entire demand of 36 by using BOP entries on operations B, D, and F, each in proportion to its routingShare: 1:3:2. Thus:

- B.execVol = 6
- D.execVol = 18
- F.execVol = 12

Initially, the heuristic attempts to meet the entire demand by using each BOP entry for a given part in proportion to its routingShare. Suppose, however, that the heuristic does not find a way to meet the entire demand while adhering to this strict proportion, due to some supply shortage or other constraint associated with a particular BOP entry. Then that BOP entry is allowed to drop out and the proportionate routing is applied only to the BOP entries that remain active. As many BOP entries may by drop out as necessary, possibly until only one is left.

Consider the following example:

FIGURE 13 Second Example of Proportionate Routing



In this case, there is not enough supply of part E to allow the BOP entry from operation F to produce its full share of the demand on part G. Thus, after operation F produces 4 units of part G, its BOP entry drops out, and the remaining demand is met by the BOP entries on operations B and D in a 1:3 ratio. Thus:

- B.execVol = 8
- D.execVol = 24
- F.execVol = 4

<u>Comments About the Proportionate Routing Technique</u>

1. To request proportionate routing for a BOM entry in a period, set the “propRtg” attribute on the BOM entry to TRUE in the period. (See “propRtg” on page 142.) In this case, the ratios are determined by the “routingShare” attribute on each of the substitutes for the BOM entry as well as the “routingShare” attribute the BOM entry itself. (See “routingShare” on page 142 and on page 147.)
2. The routingShare attributes for BOP entries, BOM entries, and substitutes are defined as vectors, which allows the initial routing ratios to be different in each period.

3. If the “propRtg” attribute is TRUE for a part or BOM entry in a period, then the multiple routes technique will not be applied to the part or BOM entry in that period.
4. If the “propRtg” attribute is TRUE for a BOM entry in a period, then the “substitute netting” logic will not be applied to the substitutes for the BOM entry in the period. See “netAllowed” on page 146. This is equivalent to setting the “netAllowed” attribute to FALSE for each of the substitutes for the BOM entry.
5. The expAversion attribute on BOP entries and expNetAversion attribute on substitutes do not apply to proportionate routing.
6. The expAllowed attributes on BOP entries and substitutes do apply to proportionate routing.

Build-Ahead

By default, heuristic implosion (and heuristic allocation) build parts “just-in-time”, i.e. the heuristic will not build parts earlier than they are needed in order to meet a demand. But this behavior can be overridden using one of the “build-ahead” options for the heuristic. There are two specific types of build-ahead:

- NSTN Build-Ahead
- ASAP Build-Ahead

NSTN Build-Ahead

NSTN (“No Sooner Than Necessary”) build-aheads applies to any material part that can be built. If a part is selected for NSTN build-ahead, then any time there is a need to produce the part, it will be built in the (heuristically) latest possible period. Thus the heuristic will first try to produce the part in the period (t) in which the part is needed. Then, if resource constraints anywhere below the part prevent the heuristic from building all of the required quantity in period t, it will try to build the remaining quantity in period t-1, and so on.

To request NSTN build-ahead for a part, set the buildNstn attribute for the part to TRUE. (See “buildNstn” on page 120.)

ASAP Build-Ahead

ASAP (“As Soon As Possible”) build-ahead works the same way as NSTN build-ahead, but in reverse. Like NSTN build-ahead, it applies to any material part that can be built. If a part is selected for ASAP build-ahead, then any time there is a need to produce the part, it will be built in the (heuristically) earliest possible period. Thus the heuristic will first try to produce the part in the earliest allowable period (which is often period 0). Then, if resource constraints anywhere below the part prevent the heuristic from building all of the required quantity in that period, it will try to build the remaining quantity in next period after that, and so on.

To request ASAP build-ahead for a part, set the buildAsap attribute for the part to TRUE. (See “buildAsap” on page 120.)

<u>The buildAheadUB Attribute</u>

The buildAheadUB attribute is a boolean attribute for each material part. (See “buildAheadUB” on page 119.) It specifies an upper bound on the number of periods by which build-ahead can be done for the part (both NSTN and ASAP). When heuristic implosion or allocation is performing build-ahead on the part in period t, the part will not be built before period t - $buildAheadUB_t$. A value of nPeriods - 1 implies no upper bound.

Stock Reallocation

(This feature only applies to heuristic implosion and allocation).

Consider the following scenario: In heuristic implosion/allocation, there is a requirement for some material part in some period, t_3, i.e, some volume of the part needs to be consumed in period t_3 in order to meet some demand. This requirement is met by consuming stock of the part that’s available starting in period $t_1 < t_3$, even though the part could have been built in period t_3. Later, a requirement for the part arrives in period t_2, where $t_1 <= t_2 < t_3$, and now there is no more stock available and the part cannot be produced in period t_2 or earlier. By default, this new requirement cannot be met on time.

The stock reallocation feature allows the period t_2 requirement to use the period t_1 stock previously allocated to the period t_3 requirement, and then produce in period t_3 to re-meet the period t_3 requirement.

To invoke the stock reallocation feature, set the global boolean attribute stockReallocation to TRUE. See “stockReallocation” on page 115.

In some cases, performing stock reallocation in the manner described above could potentially result in less of the requirement being met than not using stock reallocation at all, i.e., if more of the part could be built in period t_2 than in period t_3. To handle such cases, the stock reallocation feature uses an iterative technique that can be summarized as follows: For any given material part and requirement period (t_2), it initially performs its stock reallocations in the latest possible period. Later, in subsequent iterations, it uses progressively earlier periods until, if neccessary, it eventually gets back to using the original requirement period (t_2) and attempts to meet any remaining requirements in that period without stock reallocation.

Multiple Execution Periods

This feature only applies to heuristic implosion and allocation.

Consider an implosion problem that has BOP entries with “duplicate impactPeriods”. A BOP entry has duplicate impactPeriods if there are at least two periods, $t \neq t'$, such that

$$\text{impactPeriod}[t] = \text{impactPeriod}[t'] \geq 0$$

(See “impactPeriod” on page 150.) In other words, for at least one period in which the produced part can be built, there is more than one period in which the producing operation can be executed resulting in production in that period. This situation typically results from having an offset vector that varies by period.

The two implosion methods handle this situation differently: Optimizing implosion will use any or all of the eligible execution periods to produce the part in a given period, whatever is needed to generate an optimal solution. In contrast, heuristic implosion (and allocation) will, by default, only use one of the eligible execution periods for a given production period; specifically, it will use the last one.

The “multiple execution periods” feature allows this behavior of heuristic implosion and allocation to be overridden. To invoke the multiple execution periods feature, set the global boolean attribute “multiExec” to TRUE. (See “minimalExcess” on page 106.) When multiple execution periods is in effect, heuristic implosion and allocation will use any or all of the eligible execution periods to produce a part.

Two-Way Multiple Execution Periods

By default, the multiple execution periods technique considers the execution periods corresponding to a production period in “No-Sooner-Than-Necessary” (NTSN) order, which means that the latest period is considered first, and then the second latest, and so on. This technique is sometimes called “one-way multiple execution periods”. This behavior can be overridden by using the two-way multiple execution periods technique (two-way multi-exec). When two-way multi-exec is used, the execution periods are sometimes considered in NSTN order and sometimes in “As-Soon-As-Possible” (ASAP) order, where ASAP order means that earliest period is considered first, and then the second earliest, and so on.

The specification of whether to use NSTN ordering or ASAP ordering for a particular BOP entry and production period is called the multi-exec direction. To aid in determining which multi-exec direction is to be used in each case, an initial multi-exec direction is defined. In the case of heuristic implosion, the initial multi-exec direction is always ASAP ordering. In the case of heuristic allocation, the initial multi-exec direction is specified through API functions (see below).

For any given part, the multi-exec direction is determined by the following set of rules:

1. If the initial ASAP direction is ASAP, use ASAP ordering.
2. Otherwise, if the part uses ASAP build-ahead, use ASAP ordering.
3. Otherwise, use NSTN ordering.

To instruct heuristic implosion and allocation to use two-way multi-exec, set the global boolean attribute “twoWayMultiExec” to TRUE. (See “twoWayMultiExec” on page 116.) If twoWayMultiExec is FALSE, and multiExec is TRUE, WIT will just use one-way multi-exec.

In two-way multi-exec mode (twoWayMultiExec == TRUE), special API functions must used to invoke heuristic allocation. Specifically, instead of `witIncHeurAlloc`, the function `witIncHeurAllocTwme` must be invoked and instead of `witEqHeurAlloc`, the function `witEqHeurAllocTwme` must be invoked. These two “`Twme`” functions have an additional argument which specifies the initial multi-exec direction. In `witIncHeurAllocTwme`, a single initial multi-exec direction is specified, while in `witEqHeurAllocTwme`, an initial multi-exec direction must be specified for each allocation target.

Single-Source

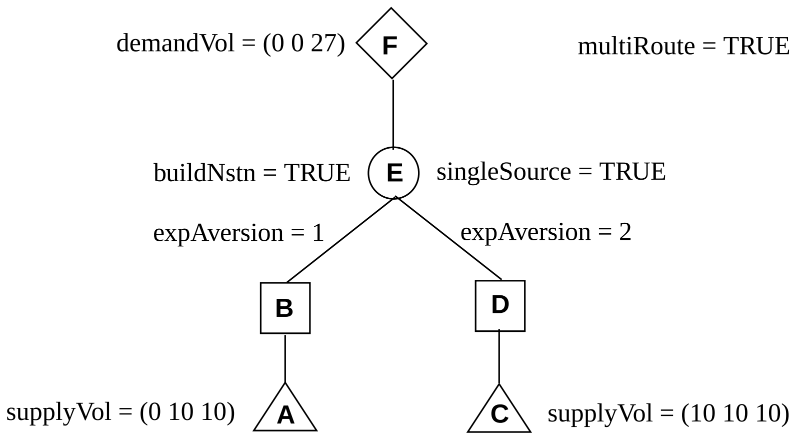
This feature only applies to heuristic implosion and heuristic allocation, and only when the multiple routes technique is being used.

The single-source technique is controlled by the singleSource boolean input attribute for parts and BOM entries. (See the singleSource attribute on page 124 and on page 142.) When singleSource is TRUE for a part, the multi-route algorithm attempts to ship the entire desIncVol specified by witIncHeurAlloc (or its equivalent in heuristic implosion) by using only one BOP entry to fill the requirements for that part. This is called “single-source” mode. The same BOP entry is used in all periods. If the heuristic fails to find a way to ship the entire desIncVol subject to this constraint, it then proceeds in “multi-source” mode for this part, using as many BOP entries for the part as needed.

When singleSource is TRUE for a BOM entry, a similar logic is applied: The multi-route algorithm attempts to ship the entire desIncVol specified by witIncHeurAlloc either by using just the BOM entry itself without substitution or by using only one of its substitutes, and switching to multi-source mode for that BOM entry only if single-source mode fails.

Consider the following example:

FIGURE 14 Example of Single-Source



If E.singleSource is set to FALSE, the resulting execution schedule is:

B.execVol = (0 7 10)
D.execVol = (0 0 10)

If E.singleSource is set to TRUE, the resulting execution schedule is:

B.execVol = (0 0 0)
D.execVol = (7 10 10)

NOTES:

1. Single-source cannot be used under any of the following conditions:
   - multiRoute = FALSE.
   - penExec = TRUE.
   - equitability > 1.
   - computeCriticalList = TRUE.
   - pgdCritListMode = TRUE
2. When the singleSource attribute is set to TRUE for a BOM entry, the heuristic's non-multi-route use of the substitutes associated with the BOM entry is shut off. The effect is the same as if the netAllowed attribute were set to FALSE for each of the substitutes associated with the BOM entry.
3. The single-source technique could potentially cause a substantial increase in the CPU time of heuristic implosion/allocation. If the increase is excessive, the run time can potentially be reduced by decreasing the number of parts and BOM entries with singleSource set to TRUE.

Penalized Execution

This feature only applies to heuristic implosion and heuristic allocation, and only when the multiple routes technique is being used.

In some applications, it may be desirable to consume resources that are higher up in the complete BOM structure before consuming those further down: The higher up resources may be expensive work-in-process inventory, while the resources further down may be cheap raw materials, etc. In problems without routing alternatives (i.e., one explodeable BOP entry for any part, and no way to build parts that are consumed by substitutes), heuristic implosion/allocation consumes resources in this manner automatically: The technique is based on explosion, which consumes resources from the top down.

However, in the presence of routing alternatives, the heuristic (by default) cannot be expected to always consume higher resources before lower ones: When the multiple routes technique is not being used, high up resources that are not on the selected routes are simply missed. When the multiple routes technique is being used, the routings are not selected on the basis of available high up resources. Furthermore, when a routing is selected, it is used until no further demand can be met on that routing, before another routing used. Thus lower resources on the first routing will be used before higher resources on the second routing.

The penalized execution technique is an extension of the multiple routes technique that attempts to consume higher resources before lower ones. To understand penalized execution, notice that “higher” versus “lower” in the complete BOM structure is not always a well-defined concept: Certainly when part A is directly below part B (so that exploding part B results in the consumption of part A), it’s clear that B is higher than A. But in other cases, the comparative “height” of two parts is not necessarily so clear. For this reason, a penalty function on the routings, called “execution penalty”, is used to define the concept of height.

In its basic form, the execution penalty of a routing is defined by a user-specified attribute for each operation: “execPenalty”. (See “execPenalty” on page 131.) This is a penalty incurred for executing the operation. Its value must be a non-negative float. The execution penalty of a routing is the sum over all operations used in the routing of:

nPaths * execPenalty

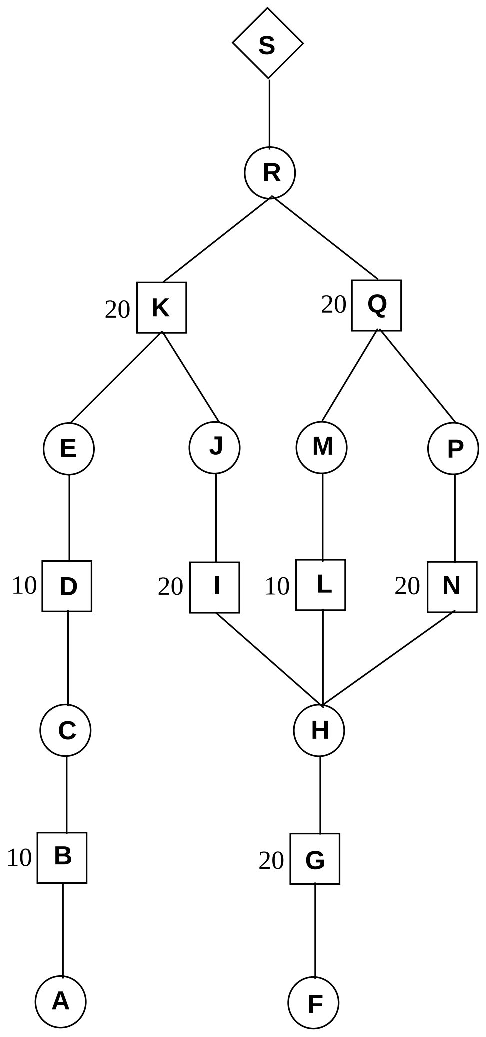
where nPaths is the number of paths in the routing by which the demand at the top of the routing uses the operation. The penalized execution technique is a heuristic for selecting a routing that has the minimum execution penalty.

The execution penalty has a number of useful properties:

- Since the individual penalties are non-negative, the cumulative penalty as one proceeds down any route can only increase or stay constant.
- The “expAversion” and “expNetAversion” attributes that are used to select routings in the ordinary multiple routes technique are used to break ties in the penalized execution technique.
- When using a particular routing, the technique may decide to use resources only “part way down” the routing before switching to another routing. This is an integral aspect of minimizing the total execution penalty of the routing, and enables the penalized execution technique to fulfill its purpose, which is to consume higher resources before lower ones.

Consider the following example:

FIGURE 15 Example of Execution Penalties



The execPenalty attribute of each operation is shown to the left of the operation. There are two main routings: one passing through operation K and terminating at parts A and F and one passing through operation Q and terminating at part F.

- The execution penalty of the “K” routing is 80: 10+10+20+20+20.
- The execution penalty of the “Q” routing is 90: (2*20)+10+20+20.

So if there is supply of parts A and F, but no supply of any other part, the “K” routing will be used before the “Q” routing.

Notice that the same operation, G, contributes 20 units of penalty to the “K” routing and 40 units of penalty to the “Q” routing, because, while there is only one path from demand S to operation G on the “K” routing, there are two paths from demand S to operation G on the “Q” routing: one each through parts M and P.

In addition to the two main routings discussed above, there are numerous partial routings extending down from demand S. In particular, if there is supply of parts part way down the BOM structure, then the routings that terminate at those parts will tend to be used first. Thus if there is supply at parts E and H, then the penalized execution technique will consider the routing through operation K terminating at parts E and H, with penalty 40, and the routing through operation Q terminating at part H, with penalty 50, and it will select the K-E-H routing first.

In some cases, it may be useful to assign a penalty to the use of BOM entries and substitutes in addition to the execution of operations. For this purpose, there is an execPenalty attribute on BOM entries and substitutes. (See “execPenalty” on page 140 and page 144.) When these penalties are being used, the execution penalty of a routing is the sum over all operations, BOM entries and substitutes used in the routing of:

nPaths * execPenalty

NOTE: If execPenalty > 0 for a BOM entry or any of its substitutes, then the netAllowed attribute must be set to FALSE for all of the substitutes for that BOM entry. (See “netAllowed” on page 146.)

To have heuristic implosion and allocation use the penalized execution technique, set the multiRoute attribute to TRUE and set the penExec attribute to TRUE. See “multiRoute” on page 107 and “penExec” on page 111.

Penalized Execution with Proportionate Routing

When penalized execution and proportionate routing are both used in the same problem, they interact in either of two distinct modes:

- Overriding Proportionate Routing, or

- Tie Breaking Proportionate Routing

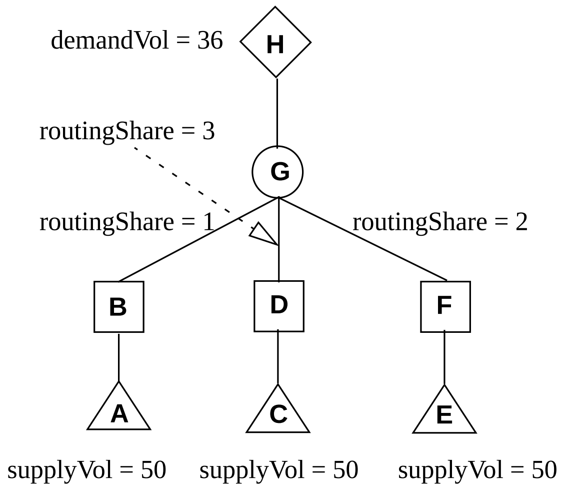
(See also “Proportionate Routing” on page 79.) The default mode is overriding proportionate routing. In this mode, proportionate routing overrides the execution penalties. Specifically, whenever the heuristic is trying to increase production of a part in a period for which the propRtg attribute is TRUE, it determines how much of the production is to be due to each BOP entry associated with the part by using the proportionate routing technique and not by using the penalized execution technique. The same overriding behavior applies to BOM entries and periods for which propRtg is TRUE.

In tie breaking proportionate routing mode, proportionate routing does not override the execution penalties; it is only used to break ties among them. Specifically, whenever the heuristic is trying to increase production of a part in a period for which the propRtg attribute is TRUE, it selects which BOP entry is to be used by employing the penalized execution technique. But if it finds two or more BOP entries that (heuristically) minimize the execution penalty, it determines how much of the production is to be due to each minimizing BOP entry by using the proportionate routing technique. The same tie breaking behavior applies to BOM entries and periods for which propRtg is TRUE.

The selection of which mode will be used for the interaction between penalized execution and proportionate routing is determined by the global boolean attribute tieBreakPropRt. (See “tieBreakPropRt” on page 115.) If tieBreakPropRt is TRUE, tie breaking proportionate routing will be used; if it is FALSE, overriding proportionate routing will be used.

Consider the following example:

FIGURE 16 Penalized Execution with Proportionate Routing



In addition, the following data apply:

Object Type	**Object ID**	**Attribute**	**Value**
Problem		nPeriods	1
Problem		multiRoute	TRUE
Problem		penExec	TRUE
Operation	B	execPenalty	1
Operation	D	execPenalty	2
Operation	F	execPenalty	1
BOP Entry	B (#0)	expAversion	2
BOP Entry	D (#0)	expAversion	0
BOP Entry	F (#0)	expAversion	1

With this example, consider the following three cases:

<u>Case 1:</u>

- `G.propRtg = FALSE`

In this case, ordinary penalized execution is applied. There is a tie for minimum execPenalty between operations B and F. The tie is resolved in favor of the smallest expAversion, so that the BOP entry from operation F is used, resulting in the following execution schedule:

- `B.execVol =  0`
- `D.execVol =  0`
- `F.execVol = 36`

<u>Case 2:</u>

- `G.propRtg      = TRUE`
- `tieBreakPropRt = FALSE`

In this case, overriding proportionate routing is applied and all three BOP entries are used in proportion to their routingShares, resulting in the following execution schedule:

- `B.execVol =  6`
- `D.execVol = 18`
- `F.execVol = 12`

<u>Case 3:</u>

- `G.propRtg      = TRUE`
- `tieBreakPropRt = TRUE`

In this case, tie breaking proportionate routing is applied. Here again, there is a tie for minimum execPenalty between operations B and F, but in this case, the tie is resolved by using proportionate routing on these two BOP entries, resulting in the following execution schedule:

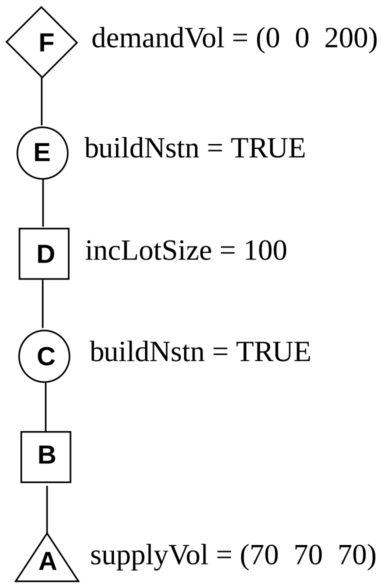
- `B.execVol = 12`
- `D.execVol =  0`
- `F.execVol = 24`

Selection Splitting

This feature only applies to heuristic implosion and allocation

Consider the following example:

FIGURE 17 Example of the Need For Selection Splitting



The following execution schedule would allow all 200 units of demandVol for demand F to be satisfied:

E.execVol = (0 0 200)

B.execVol = (60 70 70)

Unforunately, the NSTN build-ahead algorithm would (normally) build and ship 0 units for this problem, for the following reason: The selection of a build period for part C is always applied to the entire amount of part C that is needed in a given period in order to ship at least one more unit to a demand, i.e., demand F.

Since operation D always executes in multiples of 100 (due to its incLotSize), this forces part C to be built in multiples of 100 units in any given period. But since the supply of part A is only 70 units in any period, the build-ahead algorithm can’t build any units of part C in any period.

The purpose of selection splitting is to solve this kind of problem. When the build-ahead algorithm needs to select a period in which to build a part in response to a need for some quantity of the part in some period, the selection splitting technique is a heuristic for selecting more than one build period to meet this need.

To request the selection splitting technique, set the global selSplit attribute to TRUE. (See “selSplit” on page 114.) In the example above, if selSplit = TRUE, heuristic implosion produces the desireable solution described above, i.e:

E.execVol = (0 0 200)

B.execVol = (60 70 70)

Selection splitting applies to all of the following techniques:

- **Multiple Routes**: Selection splitting enables the mulitple routes technique to use more than one BOP entry in order to build a part in a period, and more than one substitute for a BOM entry in a period, corresponding to one unit of top-level demand.
- **Proportionate Routing:** Selection splitting enables the proportionate routing technique to meet one unit of top-level demand in a way that includes a given BOP entry for a portion of the prodVol of the corresponding part and excludes the BOP entry for the rest of the part’s prodVol. It applies similarly to the inclusion and exclusion of substitutes and BOM entries in proportionate routing.
- **Build-Ahead:** Selection splitting enables build-ahead to build a part in more than period in in response to a need for a quantity of the part in a single period, corresponding to one unit of top-level demand.
- **Stock Reallocation:** Selection splitting enables the requirements on a part in a single period, corresponding to one unit of top-level demand, to be met partially with stock reallocation and partially without it.
- **Multiple Execution Periods:** Selection splitting enables the multiple execution periods technique to use more than one execution period in order to build a part in a single period, corresponding to one unit of top-level demand.
- **Penalized Execution:** As with the multiple routes technique, selection splitting enables penalized execution to use more than one BOP entry in order to build a part in a period, and more than one substitute for a BOM entry in a period, corresponding to one unit of top-level demand.

NOTES:

1. Selection splitting cannot be used under either of the following conditions:
   - computeCriticalList = TRUE.
   - pgdCritListMode = TRUE
   - searchInc ≠ 1.0 for any demand.
2. The selection splitting technique can seriously increase the run time of heuristic implosion and allocation.

Selection Recovery

This feature only applies to heuristic implosion and allocation

By default, each of the heuristic's "selection" techniques (multiple routes, build-ahead, etc.) permanently discard each selection as soon as the heuristic tries to use the selection and it fails to enable an increased shipment. For example, suppose the heuristic finds that it can ship 57 units to demand B in period 6 by using NSTN build-ahead to build part A in period 4, but it cannot ship 58 units by doing so. In this case, the heuristic will go ahead and build part A in period 4 and ship the 57 units, but after it does so, it will then discard period 4 as a potential period for building part A.

Discarding failed selections improves the speed of the heuristic. In most cases, it's harmless. However, in some cases, it may cause the heuristic to miss opportunities to meet demands. For example, if the failure to enable an increased shipment by building more of a part in a period was due to a large incLotSize on the operation that was consuming the part, then there may be an opportunity to build the same part in the same period and use the produced quantity to allow execution of an operation that does not have a large incLotSize.

For cases in which the discarding of selections is harmful, the "selection recovery" capability may be used. The selection recovery capability alters the behavior of the heuristic at the end of each "allocation increment", i.e, each call to `witIncHeurAlloc`, or its internal equivalent within heuristic implosion or equitable heuristic allocation. (See "Heuristic Allocation" on page 70.) When selection recovery is in effect, at the end of each allocation increment, all discarded selections are "recovered": they are allowed to be used again in subsequent allocation increments.

To have the heuristic use selection recovery, set the global boolean attribute selectionRecovery to TRUE. (See "selectionRecovery" on page 114.)

Selection recovery applies to the following techniques:

- Multiple Routes
- Proportionate Routing
- Build-Ahead
- Stock Reallocation

- Multiple Execution Periods

Note that the use of selection recovery will tend to greatly increase the run time of the heuristic.

Lead Time Bounds

This capability only applies to heuristic implosion and allocation.

The lead time bounds capability allows an upper bound to be imposed on the time interval between the period in which a part is produced in order to enable a shipment and the period of the shipment. It is controlled by two attributes:

- leadTimeUB: A vector of integers for each demand
  (See “leadTimeUB” on page 128.)
- boundedLeadTimes: A boolean for each part
  (See “boundedLeadTimes” on page 119.)

leadTimeUB_{t} is the lead time upper bound for shipments to the demand in period t: At any time during the execution of heuristic implosion/allocation, the algorithm is seeking to increase the shipVol of one demand, ds, in one period, ts. Suppose the heuristic is considering an increase to the prodVol of a part in order to satisfy a requirement for availability of the part. Then the heuristic will only perform such an increase if it occurs in a period later than or equal to:

$\texttt{ts - leadTimeUB}_{\texttt{ts}}$

where leadTimeUB is the value of the leadTimeUB attribute of demand ds. However, this restriction only applies to parts for which boundedLeadTimes is TRUE.

Heuristic implosion will employ the lead time bounds capability, iff the boundedLeadTimes attribute is TRUE for at least one part. When this is the case, the following conditions are also required to hold:

- The global skipfailures attribute must be FALSE.
- The global selectionRecovery attribute must be TRUE.

These conditions are necessary in order to ensure proper functioning of the heuristic in the presence of lead time bounds.

The lead time bounds capability may seem similar to the buildAheadUB attribute, but it is different in a number of ways:

- It is specified by demand and shipment period.
- It applies to all aspects of the heuristic, not just build-ahead.
- It applies to both capacity and material parts.

CHAPTER 2 WIT Data Attributes

WIT has seven major types of data objects. They are: the WIT problem itself (which has global attributes), parts, demands, operations, BOM entries, substitute BOM entries, and BOP entries. Each data object has a list of attributes which fully describes the object. This chapter describes these objects and their attributes. A few of the attributes are "bound sets", which, in turn, have their own attributes. Bound set attributes are described here as well.

Many attributes are the kind whose value can be set by the user. These are called "input attributes". Other attributes are computed by WIT, and their values can be accessed by the user. These are called "output attributes". Some of the input attributes are the kind whose value is set by the user when the object is created and cannot be changed after that. These are called "immutable input attributes".

This chapter consists of a description of each attribute, organized by object type.

NOTE:

Many of the attributes are of type "float" or "vector of floats". Internally, these attributes are actually stored in double precision: either as "double" or "vector of doubles". The values of these attributes can be set or retrieved through the API either in single precision or in double precision (by calling different API functions). Also, WIT reads the values of these attributes from the input data file in double precision. For historical reasons, they are referred to as "float" attributes in this guide.

Global (WIT Problem) Attributes

accAfterOptImp

Input

Boolean

Default value: FALSE

When TRUE, this attribute tells WIT to go into in an accelerated state at the end of any optimizing implosion that is performed. See “Using Accelerated Optimizing Implosion” on page 56.

accAfterSoftLB

Input

Boolean

Default value: FALSE

When TRUE, WIT will stay in an accelerated state when any changes are made to a bound set, including changes that soften a lower bound. When FALSE, WIT will stay in an accelerated state when changes are made to a bound set, only if the changes do not soften any lower bound. See “Bound Sets and Accelerated Optimizing Implosion” on page 57.

accelerated

Output

Boolean

TRUE indicates that WIT is in an accelerated state. If optimizing implosion is invoked when this attribute is TRUE, an accelerated optimizing implosion is performed. See “The State of a WitRun” on page 161.

appData

Input

void *

Default value: NULL

“Application Data”: A pointer to any data outside of WIT that is to be associated with the WIT problem (the WitRun). For more details, see the part appData attribute on page 118.

autoPriority

Input

Boolean

Default value: FALSE

TRUE indicates that automatic priority will be used: The demand priorities will be computed during a heuristic implosion from the objective function attributes.

boundsValue

Output

Float

The value of the bounds objective. This value is meaningful only after an optimizing implosion has been performed when the data contains soft lower bounds.

compPrices

Input

Boolean

Default value: FALSE

TRUE indicates that optimizing implosion is to compute shadow prices. See “shadowPrice” on page 124.

computeCriticalList

Input

Boolean

Default value: FALSE

TRUE indicates that the implosion functions will compute the Critical Parts List. See “Critical Parts List” on page 38.

cplexEmbedded

Output

Boolean

TRUE, iff CPLEX was embedded into the current build of WIT.

cplexMipBound

Output

Float

If the CPLEX MIP solver has been invoked, cplexMipBound is the tightest upper bound that it found on the optimal objective function value. WIT obtains this value by invoking the CPLEX routine, `CPXgetbestobjval`. If cplexMipBound = objValue, then the solution has been proven optimal. If the CPLEX MIP solver has not been invoked, cplexMipBound is 0.0.

cplexMipRelGap

Output

Float

If the CPLEX MIP solver has been invoked, cplexMipRelGap is the relative gap between the objective function value for the solution and the tightest upper bound on the optimal objective function value:

```
cplexMipRelGap =
  (cplexMipBound - objValue) / (10^-10 + |cplexMipBound|)
```

WIT obtains this value by invoking the CPLEX routine, `CPXgetmiprelgap`. If cplexMipRelGap is 0.0, then the solution has been proven optimal. If the CPLEX MIP solver has not been invoked, cplexMipRelGap is -1.0.

cplexParSpecDblVal

Input

Float

Default value: 0.0

The value of a CPLEX parameter specification of type double to be created. Setting the value of this attribute has the following immediate effect: If the current value of cplexParSpecName is anything other than “NO_PARAM”, a new CPLEX parameter specification of type double is created, whose name is the value of cplexParSpecName and whose value is the specified value of cplexParSpecDblVal. If the current value of cplexParSpecName is “NO_PARAM”, no CPLEX parameter specification is created. See “CPLEX Parameter Specifications” on page 66.

cplexParSpecIntVal

Input

Int

Default value: 0.0

The value of a CPLEX parameter specification of type int to be created. Setting the value of this attribute has the following immediate effect: If the current value of cplexParSpecName is anything other than “NO_PARAM”, a new CPLEX parameter specification of type int is created, whose name is the value of cplexParSpecName and whose value is the specified value of cplexParSpecIntVal. If the current value of cplexParSpecName is “NO_PARAM”, no CPLEX parameter specification is created. See “CPLEX Parameter Specifications” on page 66.

cplexParSpecName

Input

String

Default value: “NO_PARAM”

When the value is other than “NO_PARAM”, this is the name of a CPLEX parameter specification to be created using either the cplexParSpecIntVal or cplexParSpecDblVal attributes. When the value is “NO_PARAM”, no CPLEX parameter specification is to be created based on these attributes. See “CPLEX

Parameter Specifications” on page 66.

cplexStatusCode

Output

Int

If a CPLEX solve routine has been called, cplexStatusCode is the CPLEX solution status code resulting from the most recent call. Otherwise cplexStatusCode is -1. WIT obtains the CPLEX solution status code by calling `CPXgetstat` just after calling the solve routine. For information on how to interpret CPLEX solution status codes, see the CPLEX documenation, group optim.cplex.solutionstatus. See also “CPLEX Parameter Specifications” on page 66 and “cplexStatusText” on page 101.

cplexStatusText

Output

String

If a CPLEX solve routine has been called, cplexStatusText is the string obtained from CPLEX indicating the solution status resulting from the most recent call. See also “cplexStatusCode” on page 101. For example, when cplexStatusCode is 11, cplexStatusText is “time limit exceeded”.

criticalList

Output

List of part names and periods

This is the “Critical Parts List”. See “Critical Parts List” on page 38.

currentObjective

Input

A string that matches the name of one of the objectives

Default value: "Default"

This attribute is available only in multiple objectives mode. See “Multiple Objectives Mode” on page 62. The value of this attribute is the name of the current objective, i.e., you specify the current objective by setting the value of this attribute. Whenever either of the objectiveList or objectiveListSpec attributes is set, WIT automatically sets this attribute to match the first name given in objectiveList.

currentScenario

Input

Integer ≥ 0 and < nScenarios

Default value: 0

The index of the current scenario for stochastic implosion. See “Stochastic Implosion” on page 21. This attribute is only available in stochastic mode.

equitability

Input

1 ≤ Integer ≤ 100

Default value: 1

The attribute is used by heuristic implosion and by equitable heuristic allocation. It determines the degree to which equitable allocation will be performed. If equitability is 1, equitable allocation logic is not used at all. If equitability is 100, heuristic implosion and equitable heuristic allocation will expend a maximum effort on equitable allocation. High values of equitability will result in significantly increased run times. See also “Equitable Allocation Heuristic Implosion” on page 69 and “Equitable Heuristic Allocation” on page 72.

execEmptyBom

Input

Boolean

Default value: TRUE

Consider an operation, h, and period, t, such that there are no BOM entries for operation h in effect in period t. There are two ways this situation can arise:

1. There are no BOM entries at all in the BOM of operation h, or
2. There are some BOM entries in the BOM of operation h, but none of these BOM entries is in effect during period t; i.e., for each BOM entry, k, for operation h, either t < earliestPeriod or t > latestPeriod. See “BOM Entry Attributes” on page 138 for the definitions of earliestPeriod and latestPeriod.

If execEmptyBom is TRUE, then execution of operation h in period t is allowed. If execEmptyBom is FALSE, then execution of operation h in period t is disallowed. The execEmptyBom flag applies to **all** operations h and periods t that satisfy either of the above conditions.

expCutoff

Input

Float ≥ 10^{-6}

Default value: 10^{-2}

When WIT performs an explosion as part of its computations, either for

WIT-MRP, heuristic implosion/allocation, or FSS, this computation involves the following division: When exploding through a BOP entry for a given execution period, t, WIT must divide by the effective production rate, which is productRate[t] * yieldRate[t]. Certainly, it won't do this, if the effective production rate is zero, but furthermore, it is undesireable to do this division if the effective production rate is merely close to zero, because this would have the effect of scaling up round-off errors, leading to numerical instability. To avoid this, WIT never explodes through any BOP entry and period whose effective production rate is less than the value of expCutoff.

By default, expCutoff is 10^{-2}. If you have a WIT problem in which it is necessary to explode through BOP entries with effective production rates smaller than this, you can set expCutoff to a smaller value. The disadvantage of doing so is that the resulting solution may involve more numerical "noise" than it otherwise would. (An example of numerical noise would be a solution that's slightly infeasble, because it has small negative scrapVols.)

extOptActive

Output

Boolean

TRUE, iff external optimizing implosion is currently active. See "External Optimizing Implosion" on page 59.

feasible

Output

Boolean

TRUE indicates that the current implosion solution is feasible, i.e., it satisfies all constraints. FALSE indicates that the current implosion solution is not feasible. This attribute is only valid when WIT is in a postprocessed state. See "The State of a WitRun" on page 161. If WIT is in an unpostprocessed state, this attribute is automatically FALSE.

forcedMultiEq

Input

Boolean

Default value: FALSE

Consider the situation in which equitable allocation heuristic implosion encounters a demand with a unique priority, i.e., there is no tie for priority between the demand and any other demand in the same period. Normally, WIT will allocate this demand in a single pass, avoiding the multi-pass logic of equitable allocation. (This reflects the fact that equitable allocation is not an issue in such a case.) This is the behavior if forcedMultiEq is FALSE. If forcedMultiEq is TRUE, the demand will be allocated in multiple passes, according to the "equitability" attribute, just as it would be if its priority were tied with that of another demand. (See also “Equitable Allocation Heuristic Implosion” on page 69.)

The forcedMultiEq attribute also applies to equitable heuristic allocation in the case where the list of targets contains no more than one positive desIncVol. In such a case, equitable heuristic allocation will use multiple passes if and only if forcedMultiEq isTRUE. (See also “Equitable Heuristic Allocation” on page 72.)

heurAllocActive

Output

Boolean

TRUE, if and only if heuristic allocation is active. See “Heuristic Allocation” on page 70.

highPrecisionWD

Input

Boolean

Default value: FALSE

This attribute applies when WIT is writing out the input data file. When highPrecisionWD is FALSE, values of type float are written with a moderate numerical precision. When highPrecisionWD is TRUE, values of type float are written with a higher numerical precision. For more information, see “witWriteData” on page 300.

independentOffsets

Input

Boolean

Default value: FALSE

If TRUE, the substitute BOM entry offset attribute is an input attribute and can be set independently of the offset of the corresponding BOM entry.

If FALSE, the substitute BOM entry offset attribute is an output attribute, which always has the same value as the offset of the corresponding BOM entry.

The value of this attribute can only be changed when no parts or operations have yet been created: Attempting to change its value after at least one part or operation has been created results in a severe error.

lotSizeTol

Input

Float ≥ 0.0

Default value: 10^{-5}

This attribute is used by heuristic implosion and allocation and by WIT-MRP. It is the numerical tolerance used when converting the execution volume of an operation into a lot-size feasible execution volume. (See “execVol” on page 132, “minLotSize” on page 135, and “incLotSize” on page 133.) WIT uses the following formula to determine the lot-size feasible `execVol` of an operation in a period, t:

```
execVol = mls + gridPoint * ils
```

(Except when `execVol = 0`.)

where:

```
   mls = minLotSize[t]
   ils = incLotSize[t]
```

and `gridPoint` is the location of `execVol` on the "lot-size grid".

Using exact arithmetic, `gridPoint` would be computed as:

```
gridPoint = ceil((qty - mls) / ils)
```

where `qty` is the `execVol` without lot-sizing, and `ceil(x)` is the smallest integer $\geq$ x.

Unfortunatety, since the arithmetic cannot be exact, if `qty` is already lot-size feasible, then the slightest upward error in `qty` would cause `gridPoint` to be 1 unit larger than it should be and `execVol` to be ils units larger than it should be. To avoid this situation, WIT uses the following formula:

```
gridPoint = ceil((qty - mls) / ils - lst)
```

where `lst` is the value of the `lotSizeTol` attribute. Note that this tolerance is relative to the incLotSize.

The purpose of making `lotSizeTol` an attribute is to allow the tolerance to be problem or application dependent. We recommend initially leaving it at its default value, and only changing it under the following circumstances:

- If the value of `lotSizeTol` is too small for a problem, some operations will have `execVol`s that are `incLotSize` units larger than they need to be, causing excess production. If you notice this, you might want to try setting `lotSizeTol` to a larger value.
- If the value of `lotSizeTol` is too large for a problem, some operations will have `execVol`s that are smaller than they need to be, causing slight

numerical constraint violations (≤ `lotSizeTol * incLotSize`), `which` typically appear in the solution as negative `scrapVols`. If you notice this, you might want to try setting `lotSizeTol` to a smaller value.

minimalExcess

Input

Boolean

Default value: FALSE

When minimalExcess is TRUE, each part's excessVol (in each period) is computed as the minimum portion of the part's residualVol that must be attributed to supply. When it is FALSE, the excessVol is computed as the maximum portion of the part's residualVol that can be attributed to supply. (See "excessVol" on page 121 and "residualVol" on page 123.) This attribute only affects the value of excessVol and has no effect on any other attributes in the implosion solution.

mipMode

Input

Boolean

Default value: FALSE

If mipMode is TRUE, optimizing implosion is solved as a MIP problem;

if mipMode is FALSE, optimizing implosion is solved as an LP problem.

See "MIP Mode" on page 57.

If mipMode is TRUE when optimizing implosion is invoked, the following global boolean attributes must be FALSE:

- accAfterOptImp (page 98)
- computeCriticalList (page 99)
- compPrices (page 99)

modHeurAlloc

Input

Boolean

Default value: FALSE

If modHeurAlloc is TRUE, any invocation of heuristic allocation will be performed as modifiable heuristic allocation. See "Modifiable Heuristic Allocation" on page 74.

multiExec

Input

Boolean

Default value: FALSE

If multiExec is TRUE, heuristic implosion and heuristic allocation will use the multiple execution periods technique. See “Multiple Execution Periods” on page 83.

multiObjMode

Input

Boolean

Default value: FALSE

If multiObjMode is TRUE, WIT is in multiple objectives mode; if it is FALSE, WIT is in single objective mode. See “Multiple Objectives Mode” on page 62. This attribute may only be set when no parts or operations have been created.

multiObjTol

Input

Float: $0 \leq \text{multiObjTol} \leq 10^{-3}$

Default value: 10^{-7}

This attribute is available only in multiple objectives mode. See “Multiple Objectives Mode” on page 62. This is the optimality tolerance used in multiple objectives mode. In some cases, constraining an objective to stay at its maximum value could cause numerical difficulties. To avoid such difficulties, WIT uses this numerical tolerance. When the lexicographic maximization is being performed, each prior objective is constrained to be greater than or equal to:

maxObjVal - multiObjTol * | maxObjVal |

where maxObjVal is the maximum value of the objective.

In most cases, this attribute can be left at its default value.

multiRoute

Input

Boolean

Default value: FALSE

If multiRoute is TRUE, heuristic implosion and heuristic allocation will use the multiple routes technique; if it is FALSE, they will use the single route technique. See “Multiple Routes” on page 78.

nPeriods

Input

Integer ≥ 1

Default value: 26

Number of time periods in the planning horizon.

The value of this attribute can only be changed when no parts or operations have yet been created: Attempting to change its value after at least one part or operation has been created results in a severe error.

nScenarios

Input

Integer ≥ 1

Default value: 1

Number of scenarios in stochastic implosion. See “Stochastic Implosion” on page 21.

nstnResidual

Input

Boolean

Default value: FALSE

When nstnResidual is TRUE, each part’s residualVol is computed on a “no sooner than necessary” basis: the residual volumes occur in the latest possible periods. When it is FALSE, each part’s residualVol is computed on an “as soon as possible” basis: the residual volumes occur in the earliest possible periods. This attribute only affects the values of the residualVol and excessVol attributes and has no effect on any other attributes in the implosion solution. (See “residualVol” on page 123 and “excessVol” on page 121.)

objItrState

Output

WIT-specific attribute type

This attribute identifies the current state of the object iteration process. (See “Object Iteration” on page 35.) Possible values:

- Inactive
- Located at a part
- Located at a demand
- Located at an operation
- Located at a BOM entry
- Located at a substitute
- Located at a BOP entry

objValue

Output

Float

The value of the objective function. This value is meaningful only after an optimizing implosion has been performed.

objectiveList

Input

List of Strings

Default value: {"Default"}

This attribute is available only in multiple objectives mode. See “Multiple Objectives Mode” on page 62. The value of this attribute is a list of strings, one for each objective, where each string is taken to be the name of the corresponding objective. The objective names must be distinct from each other and must not contain the character “|” (vertical bar). This attribute cannot be set from an input data file. This attribute may only be set when no parts or operations have been created. Whenever either of the objectiveList or objectiveListSpec attributes is set, WIT automatically sets the other one to match it.

objectiveListSpec

Input

String

Default value: "Default"

This attribute is available only in multiple objectives mode. See “Multiple Objectives Mode” on page 62. The value of this attribute is a single string that is taken to be the concatenation of all the objective names, with each separated by a “|” (vertical bar) character. Thus “Reward|ExecCost|OtherCost” specifies three objectives: “Reward”, “ExecCost”, and “OtherCost”. The objective names must be distinct from each other. This attribute may only be set when no parts or operations have been created. Whenever either of the objectiveList or objectiveListSpec attributes is set, WIT automatically sets the other one to match it.

objectiveSeqNo

Input

Integer: 1 ≤ objectiveSeqNo ≤ # of objectives

Default value: See below

This attribute is available only in multiple objectives mode. See “Multiple Objectives Mode” on page 62. This attribute specifies the rank ordering of the current objective in the lexicographic optimization. When optimizing implosion is invoked, each objective must have a value distinct from all the other objectives. (But note that this restriction does not apply until optimizing implosion is actually invoked.) The default value of this attribute is simply the sequence number of the objective in the objectiveList (1, 2, 3,...).

optInitMethod

Normally set by WIT, but it can also be specified as input.

Value = crash, heuristic, accelerated, or schedule

Default value: crash

Indicates that method by which the initial solution for optimizing implosion is to be generated. The value of this attribute is interpreted as follows:

• crash	CPLEX will use the dual simplex method to solve the LP problem. This method uses its own (dual) starting solution.
• heuristic	Before the optimizing implosion is performed, a heuristic implosion is performed and its solution is used as the initial solution for optimizing implosion.
• accelerated	The solution to the previous optimizing implosion is used as the initial solution for the current optimizing implosion. If optimizing implosion is performed when optInitMethod = accelerated and WIT is in an accelerated state, an accelerated optimizing implosion is performed. It is not allowed to set optInitMethod to “accelerated” when WIT is in an unaccelerated state. If this is attempted (via the API), the optInitMethod attribute is left unchanged and a warning message is issued. See “The State of a WitRun” on page 161.
• schedule	The execution and shipment schedules currently in WIT are used as the initial solution for the optimizing implosion.WIT automatically sets this attribute according to the following rules:

- It is set to “crash” by default.
- Whenever WIT is put into an accelerated state, optInitMethod is set to “accelerated”. See “The State of a WitRun” on page 161.
- Whenever WIT is changed from an accelerated state to an unaccelerated state, if the value of optInitMethod is “accelerated”, it is set to “crash”.
- Whenever the execVol, subVol, or shipVol attributes are set, optInitMethod is set to “schedule”. See “User-Specified Starting Solution for Optimizing Implosion” on page 54.

The optInitMethod attribute method can be set from the application program to any value except “accelerated”. It cannot be set from a data file.

outputPrecision

Input

Integer ≥ 0

Default value: 3

This is the number of decimal places that will be used when printing the Execution Schedule Output File and the Shipment Schedule Output File. See “Execution Schedule Output File” on page 458 of Appendix B and “Shipment Schedule Output File” on page 461 of Appendix B.

perfPegging

Input

Boolean

Default value: FALSE

Heuristic implosion and heuristic allocation will perform concurrent pegging, if and only if perfPegging is TRUE. See “Concurrent Pegging” on page 76.

periodStage

Input

Vector of integers with length equal to nPeriods.

Valid values: 0 and 1.

Default value: Vector of all 0’s

For each period, t, periodStage_t identifies the stage to which period t belongs. See “Stochastic Implosion” on page 21. This attribute is only available in stochastic mode and only in stage-by-period mode.

penExec

Input

Boolean

Default value: FALSE

Heuristic implosion and heuristic allocation will use the penalized execution technique, if and only if penExec is TRUE and multiRoute is TRUE . See “Penalized Execution” on page 87.

pgdCritList

Output

List of elements, each consisting of a critical part, a critical period, a demand, and a shipment period.

This is the “pegged critical list”. See “Pegged Criticial List” on page 78.

pgdCritListMode

Input

Boolean

Default value: FALSE

Heuristic implosion and allocation will generate a pegged critical list, if and only if this attribute is TRUE. See “Pegged Criticial List” on page 78.

pipExists

Output

Boolean

This attribute is TRUE, if and only if the post-implosion pegging has been computed and is available for retrieval. See “Post-Implosion Pegging” on page 43.

pipSeqFromHeur

Input

Boolean

Default value: FALSE

When this attribute is TRUE, the heuristic will automatically set the PIP shipment sequence to be the sequence of shipment triples that it used to construct the solution. See “Post-Implosion Pegging” on page 43.

postprocessed

Output

Boolean

TRUE indicates that WIT is in a postprocessed state. This means that postprocessing has been performed and no subsequent action has altered the input data or the execution and shipment schedules. See “The State of a WitRun” on page 161.

prefHighStockSLBs

Input

Boolean

Default value: FALSE

Applies to heuristic implosion and allocation, and only if respectStockSLBs is TRUE. If prefHighStockSLBs is TRUE, resource conflicts between soft lower bounds on the stockVols of parts higher in the multi-level BOM structure and those lower in the structure will be resolved in favor of those higher in the structure. If prefHighStockSLBs is FALSE, such conflicts will be resolved in favor of parts lower in the structure.

preprocessed

Output

Boolean

TRUE indicates that WIT is in a preprocessed state. For more information about preprocessing, see “witPreprocess” on page 320.

probability

Input

Float ≥ 0.0 and ≤ 1.0

Default value: 1/nScenarios

The probability of the current scenario in stochastic implosion. See “Stochastic Implosion” on page 21. This attribute is only available in stochastic mode.

respectStockSLBs

Input

Boolean

Default value: FALSE

Heuristic implosion and allocation will attempt to satisfy soft lower bounds on stockVols , if and only if respectStockSLBs is TRUE. See also “prefHighStockSLBs” on page 112.

roundReqVols

Input

Boolean

Default value: FALSE

If roundReqVols is TRUE, the part attributes “reqVol” and “focShortageVol” will be rounded up to the next integer. If it is FALSE, these attributes will not be rounded.

skipFailures

Input

Boolean

Default value: TRUE

If skipFailures is TRUE, heuristic implosion and allocation will, in some cases, avoid attempting to ship to a demand for a part in a period, if it previously was not able to ship all of the requested quantity to some demand for the part in the period. For typicial implosion problems, this greatly speeds up the solution time without at all degrading the solution quality (the meeting of demands). However, in some implosion problems (e.g., some problems that have operations that have multiple BOP entries), solution quality can be degraded. Setting skipFailures to FALSE overrides this behavior, so that such parts and periods will be reconsidered. This may greatly increase the CPU time consumed.

selSplit

Input

Boolean

Default value: FALSE

If TRUE, heuristic implosion and allocation will use selection splitting. See “Selection Splitting” on page 92.

selectionRecovery

Input

Boolean

Default value: FALSE

If TRUE, heuristic implosion and allocation will use selection recovery. See “Selection Recovery” on page 94.

solverLogFileName

Input

Character string

Default value: “solver.log”

Name of the file where CPLEX will write its messages.

stageByObject

Input

Boolean

Default value: FALSE

This attribute determines how stages are specified in stochastic implosion.

- If TRUE, stochastic implosion will proceed in stage-by-object mode.
- If FALSE, stochastic implosion will proceed in stage-by-period mode.

See “Stochastic Implosion” on page 21.

stochMode

Input

Boolean

Default value: FALSE

WIT is in stochastic mode, if and only if stochMode is TRUE. See “Stochastic Implosion” on page 21.

stochSolnMode

Output

Boolean

WIT is in stochastic solution mode, if and only if stochSolnMode is TRUE. See “Stochastic Implosion” on page 21. This attribute is only available in stochastic mode.

stockReallocation

Input

Boolean

Default value: FALSE

Heuristic implosion and allocation will use stock reallocation, if and only if stockReallocation is TRUE. See “Stock Reallocation” on page 83.

tieBreakPropRt

Input

Boolean

Default value: FALSE

Applies to heuristic implosion and allocation when penalized execution and proportionate routing are both being used. If tieBreakPropRt is TRUE, tie breaking proportionate routing will be used; if it is FALSE, overriding proportionate routing will be used. See “Penalized Execution with Proportionate Routing” on page 89.

title

Input

Character string

Default value: "Untitled"

The title describing the problem being modeled by WIT.

truncOffsets

Input

Boolean

Default value: FALSE

This attribute affects the interpretation of the offset attribute of all BOM entries, substitute BOM entries, and BOP entries. TRUE indicates that the offset attributes will be interpreted as “truncated”. This means that for any period, t, if offset[t] > t, WIT will act as if offset[t] = t, which implies consumption/production in period 0. If truncOffsets is FALSE, then offset[t] > t indicates consumption/production in a negative period (t - offset[t]), which implies that the operation cannot be executed in period t.

Setting truncOffsets to TRUE is probably not very useful for implosion purposes, because it would lead to schedules that are not truely feasible. Rather, the purpose of allowing truncated offsets it to enable a more full (but infeasible) MRP explosion. Normally in WIT-MRP, if a requirement is placed on a product whose producing operation has a BOM entry whose offset indicates consumption in a negative period, the product will not be produced and WIT-MRP will report a positive reqVol for this product. However, if truncated offsets are being used in this case, the product is produced, and the requirement is placed on the consumed part in period 0. In this way, the requirements are propagated all the way to the bottom of the complete BOM structure. A disadvantage to this approach is that obtaining the supplies indicated by the requirements schedule does not imply that the demands can be met on time. (It normally does imply this.) Nevertheless, many users consider truncated offsets MRP to be a useful capability.

twoWayMultiExec

Input

Boolean

Default value: FALSE

If twoWayMultiExec is TRUE, heuristic implosion and heuristic allocation will use the two-way multiple execution periods technique. See “Two-Way Multiple Execution Periods” on page 84.

NOTE: WIT ensures that whenever twoWayMultiExec is TRUE, then multiExec is also TRUE. Thus whenever twoWayMultiExec is set to TRUE, WIT sets multiExec to TRUE as well, and whenever multiExec is set to FALSE, WIT sets twoWayMultiExec to FALSE as well.

useFocusHorizons

Input

Boolean

Default value: TRUE

TRUE indicates that FSS will operate in “focus mode”, i.e., WIT will use the focus horizons to compute the FSS shipment schedule whenever a FSS is invoked. FALSE indicates that FSS will operate in “schedule mode”, i.e., the application program is expected to specify the FSS shipment schedule and WIT will not alter it.

userHeurStart

Input

Boolean

Default value: FALSE

TRUE indicates that heuristic implosion and heuristic allocation are to use the user-specified solution as their starting solution. See “User-Specified Heuristic Starting Solution” on page 75.

wbounds

Input

Float ≥ 0.0

Default value: 10000.0

The weight on the bounds objective. See “wbounds Calculation and Recommendation” on page 20.

Part Attributes

partName

Input

No default value: This attribute cannot be defaulted.

A character string which identifies the part

- Every partName must be unique.
- Every partName must have at least 1 non-blank character

partCategory

Immutable input attribute

Value = material or capacity

No default value: This attribute cannot be defaulted.

Classifies the part into one of the following categories:

• material	Any amount of the part not used in one period is available for use in the next period.
• capacity	Any amount of the part not used in one period is scrapped and is not available for use in the next period.

appData

Input

void *

Default value: NULL

“Application Data”: A pointer to some data element in the application program (if any) that is to be associated with the part. This attribute is accessible in API mode only and is intended for advanced applications. The appData attribute allows an application program to define data relevant to a part, associate this data with the part (by setting appData to a pointer to this data) and then later retrieve this data (by retrieving the appData pointer). WIT itself does nothing with the data referenced by appData. It only performs storage and retrieval of the pointer.

asapPipOrder

Input

Boolean

Default value: FALSE

If TRUE, PIP will use the ASAP pegging order on the part.

If FALSE, PIP will use the NSTN pegging order on the part.

See “Post-Implosion Pegging” on page 43 and “Pegging Order” on page 49.

Defined only for parts with partCategory = material.

belowList

Output

List of parts

This is a list of parts that are considered to be “below” the part in the complete BOM structure. This attribute was designed for WIT’s internal purposes, but it has been made accessible to the user.

As of this writing, the below list is defined as follows: Part B is in the below list of part A, if and only if there is a suitable downward path in the complete BOM structure from A to B. A “suitable downward path” is one that consists of explodeable BOP entries, any BOM entries, and any substitutes.

The below list is in downward order: If B and C are in the below list of A, and C is in the below list of B, then B appears before C in the below list of A.

Finally, note that each part is in its own below list.

boundedLeadTimes

Input

Boolean

Default value: FALSE

Lead time bounds apply to the part, iff this attribute is TRUE. See “Lead Time Bounds” on page 95.

buildAheadUB

Input

Vector of integers with length equal to nPeriods

$0 \le \text{buildAheadUB}_t \le \text{nPeriods} - 1$ For all t = 0, 1, … nPeriods - 1

Default value: nPeriods - 1

Defines an upper bound on the number of periods by which heuristic build-ahead can be done for the part. (See “Build-Ahead” on page 82.) When heuristic implosion or allocation is doing build-ahead on the part in period t, the part will not be built before period $t - \text{buildAheadUB}_t$. A value of nPeriods - 1 implies no upper bound. This attribute applies to the following forms of build-ahead:

- NSTN Build-Ahead
- ASAP Build-Ahead

This attribute is required satisfy the following self-consistency condition:

$\text{buildAheadUB}_{t+1} \leq \text{buildAheadUB}_{t} + 1$ For all t = 0, 1, … nPeriods - 2

This condition simply disallows build-ahead for period t+1 in any period earlier than is allowed for period t.

Defined only for parts with partCategory = material

buildAsap

Input

Boolean

Default value: FALSE

Applies to heuristic implosion and allocation. If buildAsap is TRUE (and the part can be built), the heuristic will use ASAP build-ahead to build the part. See “ASAP Build-Ahead” on page 82.

Defined only for parts with partCategory = material. (Build-ahead does not make sense for capacity parts.)

See also the note on page 120.

buildNstn

Input

Boolean

Default value: FALSE

Applies to heuristic implosion and allocation. If buildNstn is TRUE (and the part can be built), the heuristic will use NSTN build-ahead to build the part. See “NSTN Build-Ahead” on page 82.

Defined only for parts with partCategory = material. (Build-ahead does not make sense for capacity parts.)

NOTE: buildNstn and buildAsap are mutually exclusive boolean attributes: If buildNstn is set to TRUE for a part, then buildAsap is automatically set to FALSE for that part; if buildAsap is set to TRUE, then buildNstn is automatically set to FALSE. Of course, both attributes can be FALSE at the same time (the default state), which indicates that neither form of build-ahead is to be used for the part.

consVol

Output

Vector of floats with length equal to nPeriods.

Consumption volume for each period as determined by postprocessing. This is the amount of the part that was consumed as a result of the execution of an operation in the implosion solution. This attribute is only valid when WIT is in a postprocessed state. See “The State of a WitRun” on page 161.

excessVol

Output

Vector of floats with length equal to nPeriods

$excessVol_t \geq 0.0$ For all t = 0, 1, … nPeriods - 1

The amount that supplyVol could be reduced without making the current implosion solution infeasible. This represents the “unused supply” in each period. See also “residualVol” on page 123. This attribute is only valid when WIT is in a postprocessed state. See “The State of a WitRun” on page 161.

focusShortageVol

Output

Vector of floats with length equal to nPeriods

$focusShortageVol_t \geq 0.0$ For all t = 0, 1, … nPeriods - 1

This is the part’s contribution to the Focussed Shortage Schedule. If supplies of all parts were increased by their focusShortageVols, the FSS Shipment Schedule would be feasible. See also “roundReqVols” on page 113.

mrpConsVol

Output

Vector of floats with length equal to nPeriods.

Consumption volume for each period as determined by WIT-MRP. This is the amount of the part that was consumed according to WIT-MRP. This attribute is valid when WIT-MRP has been invoked.

mrpExcessVol

Output

Vector of floats with length equal to nPeriods

$mrpExcessVol_t \geq 0.0$ For all t = 0, 1, … nPeriods - 1

The amount that supplyVol could be reduced without altering the MRP requirements schedule. This represents the “unused supply” in each period, for MRP. See also “mrpResidualVol” on page 122. This attribute is valid when

WIT-MRP has been invoked.

mrpExcessVol is related to mrpResidualVol as follows:

$\text{mrpExcessVol}_t = \min(\text{mrpResidualVol}_t, \text{supplyVol}_t)$

mrpResidualVol

Output

Vector of floats with length equal to nPeriods

$\text{mrpResidualVol}_t \geq 0.0$ For all t = 0, 1, … nPeriods - 1

The amount by which consumption of the part could be increased without altering the MRP requirements schedule. This represents the “unused available quantity” in each period, for MRP. See also “mrpExcessVol” on page 121. This attribute is valid when WIT-MRP has been invoked.

objectStage

Input

Integer = 0 or 1

Default value: 0

Identifies the stage to which the part belongs. See “Stochastic Implosion” on page 21. This attribute is only available in stochastic mode and only in stage-by-object mode.

prodVol

Output

Vector of floats with length equal to nPeriods

$\text{prodVol}_t \geq 0.0$ For all t = 0, 1, … nPeriods - 1

Production volume (or quantity) of the part in each time period as determined by implosion. This includes the contributions of all BOP entries that indicate production of the part and takes into account operation yieldRates and BOP entry productRates.

propRtg

Input

Vector of booleans with length equal to nPeriods

Default value: FALSE in all periods

Applies to heuristic implosion and allocation. If propRtg is TRUE in a period, the heuristic will apply the proportionate routing technique to the part in that period. See “Proportionate Routing” on page 79.

reqVol

Output

Vector of floats with length equal to nPeriods

$reqVol_t \geq 0.0$ For all t = 0, 1, … nPeriods - 1

The net required quantity for each time period as determined by WIT-MRP.

This attribute for all parts comprise the Requirements Schedule. See also “roundReqVols” on page 113.

residualVol

Output

Vector of floats with length equal to nPeriods

$residualVol_t \geq 0.0$ For all t = 0, 1, … nPeriods - 1

The amount by which consumption of the part could be increased without making the current implosion solution infeasible. This represents the “unused available quantity” in each period. See also “excessVol” on page 121. This attribute is only valid when WIT is in a postprocessed state. See “The State of a WitRun” on page 161.

scrapAllowed

Input

Boolean

Default value: TRUE

Applies to optimizing implosion. If scrapAllowed if FALSE for a part, then scrapVol will be constrained to be zero in all periods in the optimizing implosion solution. (See “Disallowing Scrap” on page 62.)

scrapCost

Input

Vector of floats with length equal to nPeriods. Values may be positive, negative, or zero.

Default value: Vector of all 0.0’s

The cost incurred for each unit of the part scrapped for each time period. This is used by the objective function. Negative values are allowed, but not recommended; see “Recommendations for the Primary Objective:” on page 17

scrapVol

Output

Vector of floats with length equal to nPeriods

Amount of the part to be scrapped in each period as determined by postprocessing. This attribute is only valid when WIT is in a postprocessed state. See “The State of a WitRun” on page 161.

selForDel

Input

Boolean

Default value: FALSE

If TRUE, the user has selected this part for deletion at the next purge.

See “Object Deletion” on page 33.

shadowPrice

Output

Vector of floats with length equal to nPeriods

The vector of shadow prices for the supplyVols of the part, as computed by optimizing implosion. For any period, t, and any quantity, delta (positive or negative), shadowPrice[t] provides the following upper bound: If supplyVol[t] were changed to supplyVol[t] + delta, the resulting optimal objective function value would be ≤ its current value + delta * shadowPrice[t].

Valid only if optimizing implosion has been performed with the compPrices attribute set to TRUE, and the compPrices attribute has not been set to FALSE since then. See “compPrices” on page 99.

singleSource

Input

Boolean

Default value: FALSE

Applies to heuristic implosion and allocation. If singleSource is TRUE and the global attribute multiRoute is TRUE, the heuristic will apply the single-source technique to the part. See “Single-Source” on page 85.

stockBounds

Input

Bound set

Default value: See “Bound Set Attributes” on page 152.

Bounds on the stockVol attribute for this part.

If stocking of the part in a period is forbidden (e.g., due to a mandEC), the stockBounds for that period are ignored.

Defined only for parts with partCategory = material

stockCost

Input

Vector of floats with length equal to nPeriods. Values may be positive, negative, or zero.

Default value: Vector of all 0.0's

For each period, this value is the cost incurred for each unit of the part held in inventory at the end of the period. This is used by the objective function. Negative values are allowed, but not recommended; see "Recommendations for the Primary Objective:" on page 17

Defined only for parts with partCategory = material

stockVol

Output

Vector of floats with length equal to nPeriods

Stock (inventory) level for each period. This attribute is only valid when WIT is in a postprocessed state. See "The State of a WitRun" on page 161.

Defined only for parts with partCategory = material

supplyVol

Input

Vector of floats with length equal to nPeriods

$\text{supplyVol}_t \geq 0.0$ For all t = 0, 1, … nPeriods - 1

Default value: Vector of all 0.0's

External supplies for each time period. Also, if there is any initial inventory for the part, it should be included as supplyVol in period 0.

Demand Attributes

demandedPartName

Immutable input attribute

No default value: This attribute cannot be defaulted.

A character string which matches the partName of the demanded part

- The demanded part for a demand must be created before the demand is created.

demandName

Input

A character string which distinguishes the demand stream from all other demand streams that have the same demanded part.

No default value: This attribute cannot be defaulted.

- Every demandName must be different for each demand stream associated with a given demanded part, however, demand streams that have different demanded parts are allowed to have the same demandName. For example, this would occur if one customer is a source of demand for several different parts and you use the customer's name as the demandName.
- Every demandName must have at least 1 non-blank character

appData

Same definition as in "Part Attributes" on page 118.

cumShipBounds

Input

Bound set

Default value: See "Bound Set Attributes" on page 152.

Bounds on the cumulative shipment of part to the demand.

cumShipReward

Input

Vector of floats with length equal to nPeriods. Values may be positive, negative, or zero.

Default value: Vector of all 0.0's

For each period, t, cumShipReward_t is the per-unit reward for the total volume shipped to the demand stream during periods 0,1,...,t. This is used by the objective function.

See also "Recommendations for the Primary Objective:" on page 17.

demandVol

Input

$demandVol_t \geq 0.0$ For all t = 0, 1, … nPeriods - 1

Vector of floats with length equal to nPeriods

Default value: Vector of all 0.0’s

Demand volume for each time period. Also, if the demand has any initial backlog (from before period 0), it should be included as demandVol in period 0.

focusHorizon

Input

$-1 \leq$ Integer $<$ nPeriods

Default value: -1

Indicates the time horizon of the demand to be considered when obtaining a Focussed Shortage Schedule. A value of -1 indicates the demand is not to be considered. A value of nPeriods-1 indicates the entire demand is to be considered. A value of zero indicates only the first period demand is considered. The focusHorizon attribute is used in computing the Focussed Shortage Schedule only if the useFocusHorizons attribute is TRUE.

fssShipVol

Input

$fssShipVol_t \geq 0.0$ For all t = 0, 1, … nPeriods - 1

Vector of floats with length equal to nPeriods

Default value: shipVol (See below.)

Desired shipment volume for each time period, for Focussed Shortage Schedule (FSS) purposes. This is the demand’s contribution to the FSS Shipment Schedule. Whenever an implosion is performed, fssShipVol is automatically set to the resulting shipVol, which functions as the default value of fssShipVol.

intShipVols

Input

Boolean

Default value: FALSE

Applies to optimizing implosion only.

In MIP mode, if intShipVols is TRUE for a demand, optimizing implosion will constrain the shipVol of the demand in all periods to take on integer values only. See “MIP Mode” on page 57. When mipMode is FALSE, this attribute is ignored.

leadTimeUB

Input

Vector of integers with length equal to nPeriods.

$0 \leq$ leadTimeUB$_t$ $\leq$ nPeriods - 1 For all t = 0, 1, … nPeriods - 1

Default value: nPeriods - 1

leadTimeUB$_t$ is the lead time upper bound for shipments to the demand in period t. See “Lead Time Bounds” on page 95.

This attribute is required satisfy the following self-consistency condition:

leadTimeUB$_{t+1}$ $\leq$ leadTimeUB$_t$ + 1 For all t = 0, 1, … nPeriods - 2

This condition simply disallows parts being built for period t+1 in any period earlier than is allowed for period t.

priority

Input

Vector of integers with length equal to nPeriods.

Values may be positive, negative, or zero.

Default value: Vector of all 0’s

Priorities for the demand for each time period. For positive numbers, higher numerical values correspond to a lower priority. All priorities ≤ 0 are treated equally as the absolute lowest priority. Thus 1 is the highest priority, 2 is the second highest, and so on.

searchInc

Input

Float ≥ 0.001

Default value: 1.0

This attribute is used by heuristic implosion and heuristic allocation. When the heuristic attempts to meet demand in some period, it normally attempts to meet the entire unmet demandVol, or, in the case of heuristic allocation, to meet the entire desIncVol. However, if it is not successful in meeting the entire demand, it will attempt smaller amounts using a search technique. These smaller amounts will be multiples of the searchInc (“search increment”) for the demand.

Selection splitting cannot be used when searchInc $\neq$ 1.0. (See “Selection Splitting” on page 92.)

selForDel

Same definition as in “Part Attributes” on page 118.

shipLateAllowed

Input

Boolean

Default value: TRUE

If FALSE, then late shipments to this demand are not allowed. See “Disallowing Late Shipments” on page 37.

shipLateUB

Input

Vector of integers with length equal to nPeriods

$0 \leq$ shipLateUB$_t$ $\leq$ nPeriods - 1   For all t = 0, 1, … nPeriods - 1

Default value: nPeriods - 1

Defines an upper bound on the number of periods by which the demand can be shipped late by heuristic implosion. When considering a demandVol in period t, heuristic implosion will consider shipping the demand no later than in period t + shipLateUB$_t$. A value of nPeriods-1 implies no upper bound.

This attribute is required satisfy the following self-consistency condition:

shipLateUB$_t$ $\geq$ shipLateUB$_{t-1}$ - 1   For all t = 1, 2, … nPeriods - 1

This condition simply disallows shipping for demand in period t-1 in any period later than is allowed for period t.

shipReward

Input

Vector of floats with length equal to nPeriods. Values may be positive, negative, or zero.

Default value: Vector of all 0.0’s

For each period, t, shipReward$_t$ is the reward received for each unit shipped to the demand stream in period t. This is used by the objective function.

See also “Recommendations for the Primary Objective:” on page 17.

shipVol

Normally an output, but may be an input. See “User-Specified Solution” on page 37.

Vector of floats with length equal to nPeriods.

shipVol$_t$ ≥ 0.0

Default value: Vector of all 0.0’s

Quantity of the demanded part to be shipped to the demand stream for each time period as determined by implosion.

This attribute for all demands comprises the Shipment Schedule.

Operation Attributes

operationName

Input

No default value: This attribute cannot be defaulted.

A character string that identifies the operation

- Every operationName must be unique.
- Every operationName must have at least 1 non-blank character.

appData

Same definition as in “Part Attributes” on page 118.

execBounds

Input

Bound set

Default value: See “Bound Set Attributes” on page 152.

Bounds on execVol for the operation.

If execution of the operation in a period is forbidden (e.g., due to yieldRate = 0.0), the execBounds for that period are ignored.

execCost

Input

Vector of floats with length equal to nPeriods. Values may be positive, negative, or zero.

Default value: Vector of all 0.0’s

For each period, this value is the cost incurred for each unit of execution of the operation in this period. This is used by the objective function. Negative values are allowed, but not recommended; see “Recommendations for the Primary Objective:” on page 17

execPenalty

Input

Float ≥ 0.0

Default value: 0.0

This attribute is used by the penalized execution technique for heuristic implosion and heuristic allocation. Its value is the penalty incurred for executing the operation in a routing. See “Penalized Execution” on page 87.

executable

Output

Vector of booleans with length equal to nPeriods

$executable_t$ is TRUE if and only if it is allowable to execute the operation in period t. This is determine by WIT's preprocessing of the data prior to implosion or WIT-MRP. An operation may fail to be executable in period t for any of the following reasons:

- $yieldRate_t = 0.0$
- The offset of some BOM entry or BOP entry would result in consumption or production of a part in a period < 0 or ≥ nPeriods.
- The operation's BOM has no BOM entries in effect in period t and execEmptyBom is FALSE.

execVol

Normally, an output, but may be an input. See "User-Specified Solution" on page 37.

Vector of floats with length equal to nPeriods

$execVol_t \geq 0.0$ For all t = 0, 1, … nPeriods - 1

Default value: Vector of all 0.0's

Execution volume (or quantity) for each time period as determined by implosion. The execution volume in a period is the amount of the operation executed during the period, before yieldRate has been applied. This is given in terms of the period in which the operation completes its execution.

This attribute for all operations, along with the subVol attribute for all substitute BOM entries, comprises the Execution Schedule.

✓

NOTES:

1. By convention, the execution period for an operation is thought of as the period in which the operation **completes** execution. This convention will be literally true, if all BOM entry and BOP entry offsets for the operation are ≥ 0.0, and at least BOP entry offset = 0.0. However, there is no requirement for the user to adhere to this convention.
2. The execution volume is given in floating point format. In the real world, such volumes are usually required to be integers. However, implosion may give non-integer answers, and there are different ways to round answers into integers. To allow maximum flexibility, the execution volumes are reported without rounding. This allows the application to round in whatever way is considered suitable by the user or application developer.

fssExecVol

Output

Vector of floats with length equal to nPeriods

$fssExecVol_t \geq 0.0$ For all t = 0, 1, … nPeriods - 1

This attribute is computed as part of FSS.

This attribute for all operations, along with the fssSubVol attribute for all substitute BOM entries, comprises the FSS Execution Schedule. The FSS Execution Schedule is the execution schedule that WIT used to determine that the FSS shipment schedule could be met if the shortage schedule were added to the supply schedule.

incLotSize

Input

Vector of floats with length equal to nPeriods

$incLotSize_t \geq 0.0$ For all t = 0, 1, … nPeriods - 1

Default value: Vector of all 0.0’s

Applies during heuristic implosion and WIT-MRP. If $incLotSize_t > 0.0$, then $execVol_t$ must take one of the following values:

0.0

$minLotSize_t$

$minLotSize_t +$ $incLotSize_t$

$minLotSize_t + 2 * incLotSize_t$

$minLotSize_t + 3 * incLotSize_t$

etc.

If $incLotSize_t = 0.0$, then $execVol_t$ may take on any continuous non-negative value.

Also, if twoLevelLotSizes is TRUE, and $execVol_t \geq lotSize2Thresh_t$, then $minLotSize_t$ and $incLotSize_t$ do not apply.

incLotSize2

Input

Vector of floats with length equal to nPeriods

$incLotSize2_t \geq 0.0$ For all t = 0, 1, … nPeriods - 1

Default value: Vector of all 0.0’s

Applies during heuristic implosion and WIT-MRP. If twoLevelLotSizes is TRUE and $execVol_t \geq lotSize2Thresh_t$, then $execVol_t$ must take one of the following values:

0.0

minLotSize2$_t$

minLotSize2$_t$ + incLotSize2$_t$

minLotSize2$_t$ + 2 * incLotSize2$_t$

minLotSize2$_t$ + 3 * incLotSize2$_t$

etc.

If twoLevelLotSizes is FALSE or execVol$_t$ < lotSize2Thresh$_t$, then minLotSize2$_t$ and incLotSize2$_t$ do not apply.

intExecVols

Input

Boolean

Default value: FALSE

Applies to optimizing implosion only.

In MIP mode, if intExecVols is TRUE for an operation, optimizing implosion will constrain the execVol of the operation in all periods to take on integer values only. See “MIP Mode” on page 57. When mipMode is FALSE, this attribute is ignored.

lotSize2Thresh

Input

Vector of floats with length equal to nPeriods

lotSize2Thresh$_t$ ≥ 0.0 For all t = 0, 1, … nPeriods - 1

Default value: Vector of all 0.0’s

Applies during heuristic implosion and WIT-MRP. This attribute, along with the twoLevelLotSizes attribute, determines which lot size attributes apply.

For any period, t:

If twoLevelLotSizes is FALSE, then minLotSize$_t$ and incLotSize$_t$ are applied to execVol$_t$.

If twoLevelLotSizes is TRUE and execVol$_t$ ≥ lotSize2Thresh$_t$, then minLotSize2$_t$ and incLotSize2$_t$ are applied to execVol$_t$.

If twoLevelLotSizes is TRUE and execVol$_t$ < lotSize2Thresh$_t$, then minLotSize$_t$ and incLotSize$_t$ are applied to execVol$_t$.

minLotSize

Input

Vector of floats with length equal to nPeriods

$minLotSize_t \geq 0.0$ For all t = 0, 1, … nPeriods - 1

Default value: Vector of all 0.0's

Applies during heuristic implosion and WIT-MRP.

See "incLotSize" on page 133.

minLotSize2

Input

Vector of floats with length equal to nPeriods

$minLotSize2_t \geq 0.0$ For all t = 0, 1, … nPeriods - 1

Default value: Vector of all 0.0's

Applies during heuristic implosion and WIT-MRP.

See "incLotSize2" on page 133.

✓

NOTE: If the operation consumes a capacity part in its BOM, it is possible to set minLotSize so large that it requires more of the capacity than is available in a single period. Since the minLotSize applies to each period separately, this could result in the heuristic executing none of the operation. WIT checks for this condition and issues a warning if it is detected. We recommend any of 3 ways to alleviate this situation:

- Decrease minLotSize.
- Increase the supplyVol on the capacity part.
- Increase the length in the period. For example, if the period represents a week and minLotSize consumes 2 weeks worth of the capacity, try a period that represents 2 weeks.

mrpExecVol

Output

Vector of floats with length equal to nPeriods

$mrpExecVol_t \geq 0.0$ For all t = 0, 1, … nPeriods - 1

Execution volume for each time period as determined by WIT-MRP. The execution volume is the amount executed, before yieldRate has been applied. This attribute is valid when WIT-MRP has been invoked.

objectStage

Input

Integer = 0 or 1

Default value: 0

Identifies the stage to which the operation belongs. See “Stochastic Implosion” on page 21. This attribute is only available in stochastic mode and only in stage-by-object mode.

pipEnabled

Input

Boolean

Default value: FALSE

Specifies whether or not the operation is PIP enabled. PIP will compute operation peggings only to PIP enabled operations. See “PIP to Operations” on page 50.

pipRank

Input

Integer

Values may be positive, negative, or zero.

Default value: 0

Specifies the “rank” or “importance” of the operation in PIP to operations. See “PIP to Operations” on page 50.

selForDel

Same definition as in “Part Attributes” on page 118.

twoLevelLotSizes

Input

Boolean

Default value: FALSE

TRUE indicates that two-level lot sizes are to be used for this operation. Thus. if twoLevelLotSizes is TRUE, the lotSize2Thresh, minLotSize2 and incLotSize2 attributes for the operation may apply. If it is FALSE, these attributes do not apply. See “lotSize2Thresh” on page 134.

If twoLevelLotSizes is TRUE, the following restrictions are enforced in all periods, t:

$\text{incLotSize}_t \geq 1.0$

$\text{incLotSize2}_t \geq 1.0$

yieldRate

Input

Vector of floats with length equal to nPeriods

$0.01 \leq \text{yieldRate}_t \leq 1.0$ or $\text{yieldRate}_t = 0.0$ For all t = 0, 1, … nPeriods - 1

Default value: Vector of all 1.0s

The yield of the operation in each period. For more details on how yieldRate is applied, see “productRate” on page 151.

A yieldRate of 0.0 in period t means that the operation is not allowed to be executed in period t.

BOM Entry Attributes

✓

NOTES:

1. It is possible to have several BOM entries listing the same consumingOperationName and the same consumedPartName. Since each such BOM entry is allowed to have its own attributes, (such earliestPeriod and offset) this permits the modeling of such issues as technology changes and learning, as well as repeated use of the same part by the same operation.
2. WIT does not permit any arrangement of BOM entries, substitute BOM entries, and BOP entries that results in an explodeable cycle in the complete BOM structure. For more details, see note 2 on page 148.

consumingOperationName

Immutable input attribute

A character string which matches the operationName of the consuming operation, i.e., the operation whose Bill-Of-Manufacturing includes the BOM entry.

No default value: This attribute cannot be defaulted.

- The consuming operation for a BOM entry must be created before the BOM entry is created.

consumedPartName

Immutable input attribute

A character string which matches the partName of the consumed part, i.e., the part whose consumption is represented by the BOM entry.

No default value: This attribute cannot be defaulted.

- The consumed part for a BOM entry must be created before the BOM entry is created.

bomEntryIndex

Computed by WIT

Integer ≥ 0

An index number that distinguishes the BOM entry from all other BOM entries for the consuming operation. Specifically, it is the number of existing BOM entries for the consuming operation that were created before the current one. Thus the first BOM entry created for an operation has bomEntryIndex=0.

appData

Same definition as in “Part Attributes” on page 118.

consRate

Input

Vector of floats with length equal to nPeriods

Values may be positive, negative, or zero.

Default value: Vector of all 1.0’s

This is the number of units of the part that are consumed for each unit that the consuming operation executes, before falloutRate is applied. Thus the number of units of the part that are consumed by the execution of the operation in period t is given by the following formula:

$\text{execVol}_t * \text{consRate}_t / (1 - \text{falloutRate})$

earliestPeriod

Input

0 ≤ Integer ≤ nPeriods

Default value: 0

This is the earliest period in which the BOM entry is to be in effect. The BOM entry will only be in effect when the consuming operation is being executed in periods ≥ earliestPeriod. If earliestPeriod = nPeriods, the BOM Entry is never in effect.

☞

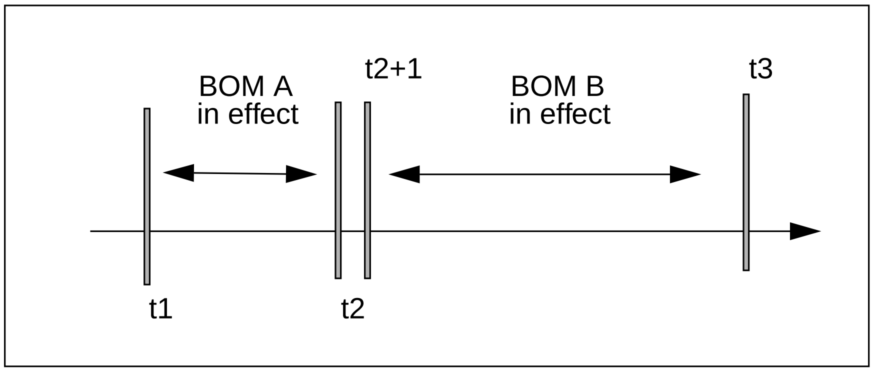
Consider the following example:

Assume that BOM entry A is valid from period t1 to period t2. Then it is replaced by BOM entry B which is valid until period t3. In this case we have to specify:

BOM-A: earliestPeriod = t1 and latestPeriod = t2

BOM-B: earliestPeriod = t2+1 and latestPeriod = t3

FIGURE 18 earliestPeriod and latestPeriod illustration



execPenalty

Input

Float ≥ 0.0

Default value: 0.0

This attribute is used by the penalized execution technique for heuristic implosion and heuristic allocation. Its value is the penalty incurred for using the BOM entry in a routing. See “Penalized Execution” on page 87.

falloutRate

Input

Float

0.0 ≤ falloutRate < 0.99

Default value: 0.0

This is the fraction of the consumed part that is lost whenever it is used by the BOM entry. For more details on how falloutRate is applied, see “consRate” on page 139.

impactPeriod

Output

Vector of integers with length equal to nPeriods.

-1 ≤ impactPeriod_t < nPeriods

Given any period t, where 0 ≤ t < nPeriods, impactPeriod_t has the following meaning:

1. If impactPeriod_t = -1, then the BOM entry is inactive in period t, i.e., it is not used when the consuming operation is executed in period t. This can occur because t < earliestPeriod or t > latestPeriod. It will also occur if the consuming operation cannot be executed in period t at all.

2. If $\text{impactPeriod}_t \geq 0$, then the BOM entry is active in period t, i.e., it is used when the consuming operation is executed in period t. In this case, the consumed part is consumed in period impactPeriod_t. Specifically,

   $\text{impactPeriod}_t \approx t - \text{offset}_t$

   If offset_t is an integer, then this expression is the exact value of impactPeriod_t.

latestPeriod

Input

$0 \leq$ Integer < nPeriods

Default value: nPeriods - 1

This is the latest effective time period for the BOM entry.

The BOM entry will only be in effect when the consuming operation is being executed in periods ≤ latestPeriod. See also “earliestPeriod” on page 139.

mandEC

Input

Boolean

Default value: FALSE

TRUE indicates that the earliestPeriod and latestPeriod attributes reflect mandatory engineering changes. If this attribute is TRUE, then periods earliestPeriod-1 and latestPeriod, are considered to be inventory purge periods for the consuming operation, reflecting a mandatory engineering change when the BOM Entry goes into effect and another mandatory engineering change when it goes out of effect. Because of the mandatory engineering change, the output of the operation is considered to be obsolete at the end of the any purge period. Thus the inventory a part is purged at the end of any period in which it is produced by an operation that is executing during one of its purge periods.

offset

Input

Vector of floats with length equal to nPeriods. Values may be positive, negative, or zero.

Default value: Vector of all 0.0’s

This is the number of periods before the period in which the consuming operation is executed that the consumed part is actually consumed. Thus if the operation is executed in period t, then the part is consumed in period $t - \text{offset}_t$. See also Note 1 on page 132.

Offsets are specified as floating point numbers rather than integers for historical reasons. It actually is recommended that offset_t be specified as an integer valued float (e.g. 1.0) for each period, t.

See also “impactPeriod” on page 140.

propRtg

Input

Vector of booleans with length equal to nPeriods

Default value: FALSE in all periods

Applies to heuristic implosion and allocation. If propRtg is TRUE in a period, the heuristic will apply the proportionate routing technique to the BOM entry in that period. See “Proportionate Routing” on page 79.

routingShare

Input

Vector of floats with length equal to nPeriods

$\text{routingShare}_t \geq 1.0$ For all t = 0, 1, … nPeriods - 1

Default value: Vector of all 1.0’s

Applies to heuristic implosion and allocation. If the proportionate routing technique is being applied to a BOM entry, then in any period, the heuristic will initially use the BOM entry and all of its substitutes, each in proportion to its routingShare for the period. See “Proportionate Routing” on page 79.

selForDel

Same definition as in “Part Attributes” on page 118.

singleSource

Input

Boolean

Default value: FALSE

Applies to heuristic implosion and allocation. If singleSource is TRUE and the global attribute multiRoute is TRUE, the heuristic will apply the single-source technique to the BOM entry. See “Single-Source” on page 85.

Substitute BOM Entry Attributes

NOTES:

1. The terms “substitute” and “substitute BOM entry” are synonymous.
2. It is possible to have several substitutes listing the same consumingOperationName, bomEntryIndex and consumedPartName. The reason for this is the same as the reason given in note 1 on page 138.
3. WIT does not permit any arrangement of BOM entries, substitute BOM entries, and BOP entries that results in an explodeable cycle in the complete BOM structure. For more details, see note 2 on page 148.

consumingOperationName

Immutable input attribute

A character string which matches the operationName of the consuming operation, i.e., the operation whose Bill-Of-Manufacturing includes the substitute.

No default value: This attribute cannot be defaulted.

- The consuming operation for a substitute BOM entry must be created before the substitute BOM entry is created.

bomEntryIndex

Immutable input attribute

Integer ≥ 0

An integer that matches the bomEntryIndex of the replaced BOM entry, i.e., the BOM entry for which the substitute BOM entry is substituting.

No default value: This attribute cannot be defaulted.

- The replaced BOM entry for a substitute BOM entry must be created before the substitute BOM entry is created.

consumedPartName

Immutable input attribute

A character string which matches the partName of the consumed part, i.e., the part whose consumption is represented by the substitute. When the substitute is being used, its “consumed part” is consumed instead of the “consumed part” of the replaced BOM entry.

No default value: This attribute cannot be defaulted.

- The consumed part for a substitute BOM entry must be created before the substitute BOM entry is created.

subsBomEntryIndex

Computed by WIT

Integer ≥ 0

An index number that distinguishes the substitute from all other substitutes for the replaced BOM entry. Specifically, it is the number of existing substitutes for the replaced BOM entry that were created before the current one. Thus the first substitute created for a BOM entry has subsBomEntryIndex=0.

appData

Same definition as in “Part Attributes” on page 118.

consRate

Same definition as in “BOM Entry Attributes” on page 138.

earliestPeriod

Same definition as in “BOM Entry Attributes” on page 138.

NOTE: A substitute BOM entry is considered to be in effect in a period only if the replaced BOM entry is also in effect. Thus a substitute BOM entry is in effect in period t, if, and only if, all of the following conditions hold:

t ≥ earliestPeriod for the substitute BOM entry

t ≥ earliestPeriod for the replaced BOM entry

t ≤ latestPeriod for the substitute BOM entry

t ≤ latestPeriod for the replaced BOM entry

execPenalty

Input

Float ≥ 0.0

Default value: 0.0

This attribute is used by the penalized execution technique for heuristic implosion and heuristic allocation. Its value is the penalty incurred for using the substitute BOM entry in a routing. See “Penalized Execution” on page 87.

expAllowed

Input

Boolean

Default value: TRUE

If TRUE, the substitute can be used for explosions. This applies only to the multiple routes technique for heuristic implosion and allocation. (See “Multiple

Routes” on page 78.) The multiple routes technique is allowed to build more quantities of the consumed part in order to be used by the substitute if and only if expAllowed is TRUE.

expNetAversion

Input

Float. Value may be positive, negative, or zero.

Default value: 0.0

An “aversion measure” of the substitute for explosion or netting. This is used by heuristic implosion and allocation. The substitutes for any given replaced BOM entry are used in order of increasing expNetAversion.

Note that substitutes with negative values of expNetAversion are treated as **preferred** substitutes for explosion and netting; those with large negative values of are preferred the most.

falloutRate

Same definition as in “BOM Entry Attributes” on page 138.

fssSubVol

Output

Vector of floats with length equal to nPeriods

$\text{fssSubVol}_t \geq 0.0$ For all t = 0, 1, … nPeriods - 1

The execution volume of the consuming operation due to the substitute as determined by FSS.

impactPeriod

Same definition as in “BOM Entry Attributes” on page 138.

intSubVols

Input

Boolean

Default value: FALSE

Applies to optimizing implosion only.

In MIP mode, if intSubVols is TRUE for a substitute, optimizing implosion will constrain the subVol of the substitute in all periods to take on integer values only. See “MIP Mode” on page 57. When mipMode is FALSE, this attribute is ignored.

latestPeriod

Same definition as in “BOM Entry Attributes” on page 138.

mrpNetAllowed

Input

Boolean

Default value: FALSE

This attribute applies to WIT-MRP and to FSS. If mrpNetAllowed is TRUE, these methods will use available supplies of the consumed part for the substitute in place of building the part consumed by the replaced BOM entry. If mrpNetAllowed is FALSE, this behavior is not allowed.

mrpSubVol

Output

Vector of floats with length equal to nPeriods

$\text{mrpSubVol}_t \geq 0.0$ For all t = 0, 1, … nPeriods - 1

The execution volume of the consuming operation due to the substitute as determined by WIT-MRP.

netAllowed

Input

Boolean

Default value: TRUE

This attribute applies to heuristic implosion and allocation. If netAllowed is TRUE, these methods will use available supplies of the consumed part for this substitute in place of building the part consumed by the replaced BOM entry. If netAllowed is FALSE, this behavior is not allowed.

offset

Input or Output, depending of the value of the global independentOffsets attribute. See also “independentOffsets” on page 104. It is an error to try to set the value of the substitute BOM entry offset attribute when independentOffsets is FALSE.

Vector of floats with length equal to nPeriods. Values may be positive, negative, or zero.

Default value: Vector of all 0.0’s

This is the number of periods before the period in which the consuming operation is executed that the consumed part is actually consumed. Thus if the operation is executed in period t, then the part is consumed in period t - offset_t. See also Note 1 on page 132.

Offsets are specified as floating point numbers rather than integers for historical reasons. It actually is recommended that offset_t be specified as an integer valued float (e.g. 1.0) for each period, t.

routingShare

Input

Vector of floats with length equal to nPeriods

$routingShare_t \geq 1.0$ For all t = 0, 1, … nPeriods - 1

Default value: Vector of all 1.0's

Applies to heuristic implosion and allocation. If the proportionate routing technique is being applied to a BOM entry, then in any period, the heuristic will initially use the BOM entry and all of its substitutes, each in proportion to its routingShare for the period. See "Proportionate Routing" on page 79.

selForDel

Same definition as in "Part Attributes" on page 118.

subCost

Input

Vector of floats with length equal to nPeriods. Values may be positive, negative, or zero.

Default value: Vector of all 0.0's

Let operation h be the consuming operation for the substitute. Then $subCost_t$ is the cost incurred for each unit of operation h that is executed in period t using the substitute. Negative values are allowed, but not recommended; see "Recommendations for the Primary Objective:" on page 17

subVol

Normally, an output, but may be an input. See "User-Specified Solution" on page 37.

Vector of floats with length equal to nPeriods.

$subVol_t \geq 0.0$ For all t = 0, 1, … nPeriods - 1

Default value: Vector of all 0.0's

The execution volume of the consuming operation due to the substitute as determined by implosion.

This attribute for all substitutes, along with the execVol attributes for all operations, comprises the Execution Schedule.

BOP Entry Attributes

NOTES:

1. It is possible to have several BOP entries listing the same producingOperationName and producedPartName. The reason for this is the same as the reason given in note 1 on page 138.
2. WIT does not permit any arrangement of BOM entries, substitute BOM entries, and BOP entries that results in an explodeable cycle in the complete BOM structure. A cycle in the complete BOM structure is a situation in which an operation either directly or indirectly consumes a part that is produced by that operation. For example, if operation AtoB consumes part A and produces part B, while operation BtoA consumes part B and produces part A, this is a cycle.

   A cycle is considered to be explodeable if and only if all of the BOP entries in it are explodeable, i.e., they have expAllowed set to TRUE. See “expAllowed” on page 149. In the above example, the cycle is explodeable if and only if the BOP entry from AtoB to B and the BOP entry from BtoA to A both have expAllowed set to TRUE.

   Explodeable cycles in the complete BOM structure are forbidden because the explosion logic used by the heuristic and WIT-MRP would theoretically be infinite if there were an explodeable cycle. To prevent this, if your data contains one or more explodeable cycles, WIT will issue a severe error message identifying one of them. It issues this error message when it performs preprocessing at the beginning of implosion or WIT-MRP. (For an API function that identifies an explodeable cycle, if any, without issuing an error message, see “witGetExpCycle” on page 315.)

producingOperationName

Immutable input attribute

A character string which matches the operationName of the producing operation, i.e., the operation whose Bill-Of-Products includes the BOP entry.

No default value: This attribute cannot be defaulted.

- The producing operation for a BOP entry must be created before the BOP entry is created.

producedPartName

Immutable input attribute

A character string which matches the partName of the produced part, i.e., the part whose production is represented by the BOP entry.

No default value: This attribute cannot be defaulted.

- The produced part for a BOP entry must be created before the BOP entry is created.

bopEntryIndex

Computed by WIT

Integer ≥ 0

An index number that distinguishes the BOP entry from all other BOP entries for the producing operation. Specifically, it is the number of existing BOP entries for the producing operation that were created before the current one. Thus the first BOP entry created for an operation has bopEntryIndex=0.

appData

Same definition as in “Part Attributes” on page 118.

earliestPeriod

Same definition as in “BOM Entry Attributes” on page 138.

expAllowed

Input

Boolean

Default value: TRUE

If TRUE, the BOP entry can be used for explosions. This applies to heuristic implosion and allocation as well as WIT-MRP and FSS. When one of these techniques needs to build more quantity of the produced part, it is allowed to use the BOP entry and its producing operation to do so, if and only if expAllowed is TRUE. If expAllowed is FALSE, the BOP entry may still be used in “by-product” mode, i.e., if the producing operation is executed for some other reason, the BOP entry will still cause the produced part to be built.

This attibute does not apply to optimizing implosion, because optimizing implosion does not use explosion logic.

See also note 2 on page 148.

expAversion

Input

Float. Value may be positive, negative, or zero.

Default value: 0.0

An “aversion measure” for exploding through the BOP entry. This applies to heuristic implosion and allocation as well as WIT-MRP and FSS. In single-route heuristic implosion and allocation as well as WIT-MRP and FSS, when more quantity of the produced part needs to be built, the eligible BOP entry with lowest expAversion is used. In multiple-routes heuristic implosion and allocation, the BOP entries associated with any given part are used in order of increasing expAversion.

Note that BOP entries with negative values of expAversion are treated as **preferred** BOP entries for explosion; those with large negative values of are preferred the most.

This attibute does not apply to optimizing implosion, because optimizing implosion does not use explosion logic.

impactPeriod

Output

Vector of integers with length equal to nPeriods.

$-1 \leq \text{impactPeriod}_t < \text{nPeriods}$

Given any period t, where $0 \leq t < \text{nPeriods}$, impactPeriod_t has the following meaning:

1. If $\text{impactPeriod}_t = -1$, then the BOP entry is inactive in period t, i.e., it is not used when the producing operation is executed in period t.
2. If $\text{impactPeriod}_t \geq 0$, then the BOP entry is active in period t, i.e., it is used when the producing operation is executed in period t. In this case, the produced part is produced in period impactPeriod_t.

   See also BOM entry impactPeriod on page 140.

latestPeriod

Same definition as in “BOM Entry Attributes” on page 138.

offset

Input

Vector of floats with length equal to nPeriods. Values may be positive, negative, or zero.

Default value: Vector of all 0.0’s

This is the number of periods before the period in which the producing operation is executed that the produced part is actually produced. Thus if the operation is executed in period t, then the part is produced in period $t - \text{offset}_t$. See also Note 1 on page 132.

Offsets are specified as floating point numbers rather than integers for historical reasons. It actually is recommended that offset_t be specified as an integer valued float (e.g. 1.0) for each period, t.

See also “offset” on page 141. See also “impactPeriod” on page 150.

productRate

Input

Vector of floats with length equal to nPeriods

$\text{productRate}_t \geq 0.0$ For all t = 0, 1, … nPeriods - 1

Default value: Vector of all 1.0’s

This is the number of units of the part that are produced for each unit executed of the producing operation, after yieldRate is applied. Thus the number of units of the part that are produced by the execution of the operation in period t is given by the following formula:

$\text{execVol}_t * \text{productRate}_t * \text{yieldRate}_t$

routingShare

Input

Vector of floats with length equal to nPeriods

$\text{routingShare}_t \geq 1.0$ For all t = 0, 1, … nPeriods - 1

Default value: Vector of all 1.0’s

Applies to heuristic implosion and allocation. If the proportionate routing technique is being applied to a part, then in any period, the part will initially be produced using all of its BOP entries, each in proportion to its routingShare for the period. See “Proportionate Routing” on page 79.

selForDel

Same definition as in “Part Attributes” on page 118.

Bound Set Attributes

A bound set is used to define the allowed range of values in each period. It consists of three vectors. They are:

hardLowerBound

vector of floats with length nPeriods

$\text{hardLowerBound}_t \geq 0.0$ for all $t = 0,1,...,\text{nPeriods}-1$

Default value: Vector of all 0.0’s

The mandatory lower bound on an attribute in period t is hardLowerBound_t. If this bound cannot be satisfied, the problem is found to be infeasible.

softLowerBound

vector of floats with length nPeriods

$\text{softLowerBound}_t \geq 0.0$ for all $t = 0,1,...,\text{nPeriods}-1$

Default value: Vector of all 0.0’s

The desired lower bound on an attribute in period t is softLowerBound_t. Violations of the soft lower bound are permitted, but penalized.

hardUpperBound

vector of floats with length nPeriods

May be positive, zero, or negative

Default value: Vector of all -1.0’s

The mandatory upper bound on an attribute in period t is hardUpperBound_t. The LP model is formulated in such a way that this kind of bound can always be satisfied.

A negative hardUpperBound is interpreted to mean $+\infty$.

An upper bound of $+\infty$ indicates that the value does not have an upper bound and can be as large as needed.

An additional requirement is:

$\text{hardLowerBound}_t \leq \text{softLowerBound}_t \leq \text{hardUpperBound}_t$

for all t = 0,1,...,nPeriods-1

(where for hardUpperBound, negative numbers are interpreted as $+\infty$)

CHAPTER 3

Using the WIT Stand-Alone Executable

How to Run the WIT Stand-Alone Executable

The WIT Stand-Alone Executable communicates primarily through flat disk files. (The program also displays some messages on the screen when it's running.) The files are described in Table 5 on page 154. Each file has a default name, which can be overridden. The default name for the Control Parameter file can be overridden by specifying it as the command line argument to WIT. The default name for all other files can be overridden by specifying them in the Control Parameter File. See “Control Parameter File” on page 452 of Appendix B.

The default file names are platform-dependent. See “Default File Names for the Stand-Alone Executable on AIX” on page vi.

The executable file for WIT is called `wit`. If the name of your Control Parameter file is <`name`>, type `wit` <`name`> to run the WIT Stand-Alone Executable.

TABLE 5 Input and Output File Descriptions (Stand-Alone Executable)

File	Direction	Control Parameter	Content
Control Parameter File	Input	none, see page 153.	Control parameters
Input Data File	Input	data_ifname	Data that defines the WIT problem to be solved.
Status Log File	Output	log_ofname	Log of various status messages. Also includes error messages. When implosion by optimization is used, the optimal objective values are reported here.
Echo Output File	Output	echo_ofname	Optional output of input data just after it was read. (Useful for verifying that WIT has read your data correctly.)
Pre-processing Output File	Output	pre_ofname	Optional output of data after various pre-processing has been performed.
LP Solver Log File	Output	solver_ofname	Messages from the LP solver.
Execution Schedule Output File	Output	exec_ofname	Optional output of the Execution Schedule
Shipment Schedule Output File	Output	ship_ofname	Optional output of the Shipment Schedule
Requirements Schedule Output File	Output	mrpsup_ofname	Optional output of the Requirements Schedule.
Comprehensive Implosion Solution Output File	Output	soln_ofname	Optional output of the Comprehensive Implosion Solution. This consists of the Execution Schedule, Shipment Schedule, Focussed Shortage Schedule with universal focus, and other related schedules.
Comprehensive MRP Solution Output File	Output	soln_ofname	Optional output of the Comprehensive MRP Solution. This consists of the MRP Requirements Schedule and other related schedules.
Critical Parts List Output File	Output	critical_ofname	Optional output of the Critical Parts List.

CHAPTER 4

Using the API: Application Program Interface

API Data Types

Real numbers passed to WIT are of the type `float` or the type `double`. Integer numbers are typically of the type `int`. The function definition provides the correct type. Many WIT functions pass or return vectors. These vectors always have length equal to nPeriods or the vector length is a parameter. Data types specific to WIT are defined in the file `wit.h`. They are:

- `WitRun`

  A structure which defines the WIT problem. A pointer to this structure is obtained by using the function `witNewRun` and is the first parameter of each WIT API function.
- `witBoolean`

  The constants `WitTRUE` and `WitFALSE` are parameters to several WIT functions having the type `witBoolean`. `WitTRUE` and `WitFALSE` are defined in the file `wit.h`.
- `witAttr`

  This type is used to define several constants passed to WIT functions. These constants are defined in `wit.h`.
- `witReturnCode`

  WIT function return codes are of the type `witReturnCode` and are either:

  ```
  WitINFORMATIONAL_RC
  WitWARNING_RC
  WitSEVERE_RC, or
  WitFATAL_RC.
  ```

  The return code represents the highest severity message condition which occurred during the function invocation. The definitions `WitINFORMATIONAL_RC`, `WitWARNING_RC`, `WitSEVERE_RC`, and `WitFATAL_RC` are in the file `wit.h`.

  Since the relation

```
WitINFORMATIONAL_RC< WitWARNING_RC< WitSEVERE_RC< WitFATAL_RC
```

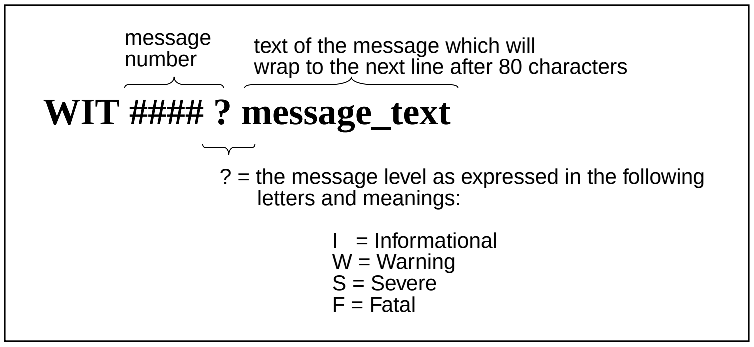
  is true, an application program can check to see if the return code is greater or equal to `WitSEVERE_RC` to check for a severe or fatal return code.

- `WitErrorExc`

  This type is a C++ class that will be thrown as an exception if an error message is issued for which the `mesgThrowErrorExc` attribute is TRUE. (See “WIT Error Exceptions” on page 158.)

API Message Attributes

WIT messages are of the form:


message number
text of the message which will wrap to the next line after 80 characters
WIT #### ? message_text
? = the message level as expressed in the following letters and meanings:
I = Informational
W = Warning
S = Severe
F = Fatal

Message levels including the following detailed meanings:

- **I** is informational. These messages provide information on what WIT is doing.
- **W** is warning. These messages indicate that WIT has recognized a situation which may not be the user's intention.
- **S** is severe. Severe messages indicate that something has occurred that caused the WIT API function to terminate prematurely. By default, WIT terminates execution of the application immediately after issuing a severe message.

  However, severe messages that result from setting part, demand, BOM entry, or substitute BOM entry attributes to out-of-range values do not cause WIT to terminate execution of the application immediately. In these cases, execution is terminated when a function is invoked that causes WIT to preprocess that data: This includes `witPreprocess`, `witHeurImplode`, `witOptImplode`, etc. This allows the application to identify all out-of-range values with a single run. This is called the “delayed termination case”.

  The default action of terminating the application after issuing a severe message can be altered by changing the `mesgStopRunning` attribute. If the `mesgStopRunning` attribute is false, then the WIT routine issuing the message immediately returns to the application with a return code of `WitSEVERE_RC`, or it throws an exception.
- **F** is fatal. Fatal messages indicate that WIT has recognized a condition which probably indicates a WIT internal programming error.

mesgFileAccessMode

`char *`

Default value: "w"

This is the file access mode WIT uses when opening files with the C function `fopen`. For more information, see the ANSI C `fopen` function.

mesgFile

`FILE *`

This file is used to write WIT messages.

mesgFileName

`char *`

Default value: `WitSTDOUT`

Name of file where WIT writes messages. Either of the values `WitSTDOUT` or `"stdout"` can be used to indicate that messages are to be written to `stdout`.

The acceptable values depends on the platform. Since the C function `fopen` is used to open the file, see `fopen` documentation for the platform being used.

mesgPrintNumber

`witBoolean`

Default value: `WitTRUE`

Associated with individual messages. If set to `WitTRUE`, the message is displayed the with WIT####? message number.

mesgStopRunning

`witBoolean`

Default value: `WitTRUE`

This attribute is associated with individual messages. It can be set and retrieved for any message, but it only applies if the message is of level “severe” or “fatal”. If set to `WitTRUE`, WIT will immediately terminate execution of the application program after issuing the message. (However, see also the “delayed termination case” on page 156.) When `mesgStopRunning` is set to `WitTRUE`, WIT automatically sets `mesgThrowErrorExc` to `WitFALSE`.

mesgThrowErrorExc

`witBoolean`

Default value: `WitFALSE`

This attribute is associated with individual messages. It can be set and retrieved for any message, but it only applies if the message is of level “severe” or “fatal”. If set to `WitTRUE`, WIT will throw an exception of type `WitErrorExc` after issuing the message. (However, see also the “delayed termination case” on page 156.) For information on `WitErrorExcs`, see “WIT Error Exceptions” on page 158. When `mesgThrowErrorExc` is set to `WitTRUE`, WIT automatically sets `mesgStopRunning` to `WitFALSE`.

NOTE:
If `mesgStopRunning` and `mesgThrowErrorExc` are both `WitFALSE` for a particular message, and the message is of level “severe” or “fatal”, then after WIT issues the message, the API function that was called by the application program will immediately return execution to the application program with a return code of either `WitSEVERE_RC` or `WitFATAL_RC`. (However, see also the “delayed termination case” on page 156.). See also “Error Recovery” on page 160.

mesgTimesPrint

`0` ≤ Integer ≤ `UCHAR_MAX`

Default value depends on the message.

This attribute is associated with individual messages. It indicates how many times a message is printed. `UCHAR_MAX` defined in the file `limits.h` indicates that the message will always be displayed. Zero indicates that the message will never be displayed.

WIT Error Exceptions

If the application program requests it, WIT can use the “exception” feature of C++ to indicate error conditions. Note, however, that exception handling is not available in C, so this capability cannot be used within application programs written in C. If the `mesgThrowErrorExc` attribute is `WitTRUE` for a particular message, and the message is of level “severe” or “fatal”, then after WIT issues the message, it will throw an exception of type `WitErrorExc`. If `mesgThrowErrorExc` has been set to `WitTRUE` for any message, it is the application program’s responsibility to catch exceptions of type `WitErrorExc`. The exception type `WitErrorExc` is a C++ class declared as follows in `wit.h`:

```
class WIT_DECLSPEC WitErrorExc
   {
   public:

      WitErrorExc (const char *, int, witReturnCode);

      WitErrorExc (const WitErrorExc &);

     ~WitErrorExc ();

      const char *  funcName () const;
      int           mesgNum  () const;
      witReturnCode retCode  () const;

   private:

      char                funcName_[51];
      const int           mesgNum_;
      const witReturnCode retCode_;
   };
```

The public member functions of this class have the following interpretation:

`funcName ()`

Returns the name of the WIT API function that threw the error exception.

`mesgNum ()`

Returns the message number of the message that caused the error exception to be thrown.

`retCode ()`

Returns the return code that the WIT API function would have returned if the mesgThrowErrorExc attribute had been WitFALSE for the error message that triggered the error exception. Its value will be either `WitSEVERE_RC`, or `WitFATAL_RC`.

See also “Error Recovery” on page 160.

For an example of a short application program that catches an error exception thrown by WIT, see “Sample 4: checkData.C” on page 429.

Error Recovery

If a WIT API function either returns with a return code ≥ `WitSEVERE_RC` or throws a WitErrorExc, this indicates that WIT has recovered from a severe or fatal error condition. When this has occurred, the following restrictions apply:

- The WitRun that was used as an argument to the API function must not be used in any further API function calls.
- If the error occurred during a call to `witCopyData`, then the above restriction applies to both WitRun arguments.
- If the error occurred during a call to `witReadData`, then no further calls to `witReadData` may be made, using any WitRun.

The C++ `std::set_new_handler` Function

The following pertains only to application programs written in C++. By default, when the C++ “`new`” operator is unable to allocate the requested memory, a `std::bad_alloc` exception is thrown. It is possible to override this behavior by using the function `std::set_new_handler`. However, WIT’s proper functioning depends on this default “`bad_alloc`” behavior, and so therefore it is important that any WIT application program written in C++ must either not use `std::set_new_handler` at all or must use it in a way that preserves the default “`bad_alloc`” behavior.

Thread Safety

The WIT API can be used safely in a multiply-threaded environment such as Java. When invoking WIT API functions in a multiply-threaded environment, the following considerations should be observed:

- Two calls to WIT API functions are considered to be “overlapping”, if the second call is initiated before the previous call is concluded.
- Overlapping calls to API functions with the same WitRun argument <u>are</u> <u>not</u> allowed. (Note that this restriction applies to both WitRun arguments of `witCopyData`.)
- Overlapping calls to `witReadData` with different WitRun arguments <u>are</u> <u>not</u> allowed.
- Overlapping calls to API functions with different WitRun arguments <u>are</u> allowed, if neither call or only one of the calls is to `witReadData`.

If any of the above restrictions is violated, WIT will usually issue a severe error message. Unfortunately, in a multiply-threaded environment, WIT cannot always detect violations of the “function call overlap” restrictions. For example, if two overlapping calls are made to API functions with the same WitRun argument, there is a small chance that WIT will not detect this occurance. In such a case, WIT’s behavior is undefined.

API Bound Set Definition

The term bound set describes three ordered float vectors which describe a boundary condition. When using the API each vector has length equal to the number of time periods. The vectors are:

- Hard lower bounds
- Soft lower bounds
- Hard upper bounds

When a bound set is passed to a WIT function, these 3 ordered vectors are passed. If one of the vectors is `NULL` then that vector is unchanged. For more information see "Bound Set Attributes" on page 152.

The State of a WitRun

At any time, a given WitRun is considered to be in some "state". The state of a WitRun determines which of the WitRun's internal data structures are currently valid. It is determined by the sequence of API calls that have been previously made for the WitRun, and in some cases, it influences the effect that the next API call will have. The state of a WitRun is characterized by the following two boolean attributes:

accelerated

True if and only if an optimizing implosion has been performed while accAfterOptImp was True and all subsequent actions were compatible with an accelerated state. If this attribute is True, then the WitRun is considered to be in an accelerated state. If it is False, the Witrun is considered to be in an unaccelerated state. When a WitRun is in an accelerated state, the data structures that are necessary in order to perform an accelerated optimizing implosion exist and are in a valid state. When an application calls `witOptImplode`, the resulting optimizing implosion will be an accelerated optimizing implosion if and only if the WitRun is in an accelerated state and the optInitMethod attribute = "accelerated". The only way to put a WitRun into an accelerated state is to call `witOptImplode` when the accAfterOptImp attribute is True. Various functions will put the WitRun into an unaccelerated state; see Table 6 on page 162.

postprocessed

True if and only if postprocessing has been performed and no subsequent action has altered the input data or the production and shipment schedules. If this attribute is True, then the WitRun is considered to be in a postprocessed state. If it is False, the WitRun is considered to be in an unpostprocessed state. Postprocessing is automatically performed at the end of the implosion routines. It computes the following data attributes:

- feasible
- stockVol
- scrapVol
- excessVol

When a WitRun is in a postprocessed state, these attributes are in a valid state in the sense that they correspond to the current input data and current production and shipment schedules. In particular, it is an error to call `witGetFocusShortageVol` or `witGetPartFocusShortageVol` when the WitRun is in an unpostprocessed state, because these attributes must be valid in order for WIT to compute a focussed shortage schedule; see “Focussed Shortage Schedule” on page 40. The following functions put the WitRun into a postprocessed state:

- `witHeurImplode`
- `witOptImplode`
- `witPostprocess`

Any function that changes the definition of the implosion problem or changes the production and shipment schedules will put the WitRun into an unpostprocessed state; see Chapter 5, “API Function Library”.

General Comments about State Attributes

The state attributes, accelerated, and postprocessed, are considered to be global data attributes of WIT (See “Global (WIT Problem) Attributes” on page 98.) and their values can be obtained by the appropriate API “get” routine. (See “witGetAttribute” on page 169.) Also, when the value of any state attribute changes, a message is displayed.

To determine which attributes can be changed while preserving an accelerated state, see Table 6 on page 162. The attributes corresponding to a “No” in the right-hand cannot be changed while preserving an accelerated state. For example, if you call the function `witSetOperationYieid` on a WitRun in an accelerated state, the WitRun will be put into an unaccelerated state. But if you call the function `witSetPartSupplyVol` on a WitRun in an accelerated state, the WitRun will remain in an accelerated state.

TABLE 6 **Which Attributes Can Be Changed While Preserving an Accelerated State**

Attribute (Input attributes only)	Object Type	Can this attribute be changed while preserving an accelerated state?
accAfterOptImp	Global	Setting it to True preserves an accelerated state. Setting it to False puts the WitRun in an unaccelerated state.
accAfterSoftLB	Global	No

TABLE 6 **Which Attributes Can Be Changed While Preserving an Accelerated State**

Attribute (Input attributes only)	Object Type	Can this attribute be changed while preserving an accelerated state?
appData	Any	Yes
asapPipOrder	Part	Yes
autoPriority	Global	Yes
boundedLeadTimes	Part	Yes
buildAheadUB	Part	Yes
buildAsap	Part	Yes
buildNstn	Part	Yes
compPrices	Global	Yes
computeCriticalList	Global	Yes
consRate	BOM Entry Substitute BOM Entry	No
cplexParSpecDblVal	Global	Yes
cplexParSpecIntVal	Global	Yes
cplexParSpecName	Global	Yes
cumShipBounds	Demand	See “Bound Sets and Accelerated Optimizing Implosion” on page 57.
cumShipReward	Demand	Yes
currentScenario	Global	No
demandName	Demand	Yes
demandVol	Demand	Yes
earliestPeriod	BOM Entry Substitute BOM Entry BOP Entry	No
equitability	Global	Yes
execBounds	Operation	See “Bound Sets and Accelerated Optimizing Implosion” on page 57
execCost	Operation	Yes
execEmptyBom	Global	No
execPenalty	Operation BOM Entry Substitute BOM Entry	Yes
execVol	Operation	No
expAllowed	Substitute BOM Entry BOP Entry	No
expAversion	BOP Entry	No
expCutoff	Global	No
expNetAversion	Substitute BOM Entry	Yes
falloutRate	BOM Entry Substitute BOM Entry	No

TABLE 6 **Which Attributes Can Be Changed While Preserving an Accelerated State**

Attribute (Input attributes only)	Object Type	Can this attribute be changed while preserving an accelerated state?
focusHorizon	Demand	Yes
forcedMultiEq	Global	Yes
highPrecisionWD	Global	Yes
incLotSize	Operation	No
incLotSize2	Operation	No
independentOffsets	Global	No
intExecVols	Operation	No
intShipVols	Demand	No
intSubVols	Substitute BOM Entry	No
latestPeriodt	BOM Entry Substitute BOM Entry BOP Entry	No
leadTimeBounds	Demand	Yes
lotSize2Thresh	Operation	No
lotSizeTol	Global	Yes
mandEC	BOM Entry Substitute BOM Entry	No
minLotSize	Operation	No
minLotSize2	Operation	No
minimalExcess	Global	Yes
mipMode	Global	No
modHeurAlloc	Global	Yes
mrpNetAllowed	Substitute BOM Entry	Yes
multiExec	Global	No
multiRoute	Global	No
nPeriods	Global	No
nScenarios	Global	Yes
netAllowed	Substitute BOM Entry	Yes
nstnResidual	Global	Yes
objectStage	Part Operation	No
offset	BOM Entry BOP Entry Substitute BOM Entry	No
operationName	Operation	Yes
optInitMethod	Global	No
partName	Part	Yes
penExec	Global	Yes

TABLE 6 **Which Attributes Can Be Changed While Preserving an Accelerated State**

Attribute (Input attributes only)	Object Type	Can this attribute be changed while preserving an accelerated state?
perfPegging	Global	Yes
periodStage	Global	No
pgdCritListMode	Global	Yes
pipEnabled	Operation	Yes
pipRank	Operation	Yes
pipSeqFromHeur	Global	Yes
prefHighStockSLBs	Global	Yes
priority	Demand	Yes
probability	Global	No
productRate	BOP Entry	No
propRtg	Part BOM Entry	Yes
respectStockSLBs	Global	Yes
roundReqVols	Global	Yes
routingShare	BOM Entry Substitute BOM Entry BOP Entry	Yes
scrapAllowed	Part	No
scrapCost	Part	Yes
searchInc	Demand	Yes
selForDel	Any	Yes
selSplit	Global	Yes
selectionRecovery	Global	Yes
shipLateAllowed	Demand	No
shipLateUB	Demand	Yes
shipReward	Demand	Yes
shipVol	Demand	No
singleSource	Part BOM Entry	Yes
skipFailures	Global	Yes
solverLogFileName	Global	Yes
stageByObject	Global	Yes
stochMode	Global	No
stockBounds	Part	See “Bound Sets and Accelerated Optimizing Implosion” on page 57
stockCost	Part	Yes
stockReallocation	Global	Yes
subCost	Substitute BOM Entry	Yes
subVol	Substitute BOM Entry	No

TABLE 6 Which Attributes Can Be Changed While Preserving an Accelerated State

Attribute (Input attributes only)	Object Type	Can this attribute be changed while preserving an accelerated state?
supplyVol	Part	Yes
tieBreakPropRt	Global	Yes
title	Global	Yes
truncOffsets	Global	No
twoLevelLotSizes	Operation	No
twoWayMultiExec	Global	No
useFocusHorizons	Global	Yes
userHeurStart	Global	Yes
wbounds	Global	Yes
yieldRate	Operation	No

CHAPTER 5 API Function Library

The WIT API Function Library consists of C functions to:

- Access and modify global data
- Access and modify part data
- Access and modify demand data
- Access and modify operation data
- Access and modify BOM entry data
- Access and modify substitute BOM entry data
- Access and modify BOP entry data
- Perform major actions such as implosion or explosion
- Read and write files
- Perform utilities
- Copy input data from one data object into another data object.
- Perform object iteration
- Work with post-implosion pegging
- Control messages
- Access and modify data using double precision.
- Perform external optimizing implosion

API typedefs, defines, constants, and function prototypes are in the file `wit.h`. Any application program invoking a WIT API function should include the file `wit.h`.

General Error Conditions

Most WIT API functions detect and report various errors in their use. Many of the error conditions are specific to the API function in question and are described in this chapter under the function in question. However, the following error conditions apply to all WIT API functions, except those whose description specifically indicates otherwise:

- A call to a given function with a given WitRun must be preceeded by a call to `witInitialize` with the same WitRun.
- A pointer argument must not be a NULL pointer.
- If a function either returns with a return code ≥ `WitSEVERE_RC` or throws a WitErrorExc, then the WitRun that was used as an argument to the function must not be used in any further API function calls.

Global Data Functions

The following functions allow the application program to access and modify data associated with an entire WitRun., either global attributes (e.g., nPeriods) or collections of data objects (e.g., the set of all parts).

witGetAttribute

```
witReturnCode witGetAttribute
(    WitRun * const theWitRun,
     Type value );
```

`witGetAttribute` represents a group of functions for obtaining the value of global attributes. For more information on global attributes see “Global (WIT Problem) Attributes” on page 98.

The `witGetAttribute` functions are:

```
witReturnCode witGetAccAfterOptImp
(    WitRun * const theWitRun,
     witBoolean * accAfterOptImp );
witReturnCode witGetAccAfterSoftLB
(    WitRun * const theWitRun,
     witBoolean * accAfterSoftLB );
witReturnCode witGetAccelerated
(    WitRun * const theWitRun,
     witBoolean * accelerated );
witReturnCode witGetAppData
(    WitRun * const theWitRun,
     void * * appData );
witReturnCode witGetAutoPriority
(    WitRun * const theWitRun,
     witBoolean * autoPriority );
witReturnCode witGetBoundsValue
(    WitRun * const theWitRun,
     witBoolean * boundsValue );
witReturnCode witGetCompPrices
(    WitRun * const theWitRun,
     witBoolean * compPrices );
witReturnCode witGetComputeCriticalList
(    WitRun * const theWitRun,
     witBoolean * computeCriticalList );
witReturnCode witGetCplexEmbedded
(    WitRun * const theWitRun,
     witBoolean *   cplexEmbedded);
witReturnCode witGetCplexMipBound
```

```
(    WitRun * const theWitRun,
     float *        cplexMipBound);
witReturnCode witGetCplexMipRelGap
(    WitRun * const theWitRun,
     float *        cplexMipRelGap);
witReturnCode witGetCplexParSpecDblVal
(    WitRun * const theWitRun,
     float *        cplexParSpecDblVal);
witReturnCode witGetCplexParSpecIntVal
(    WitRun * const theWitRun,
     int *          cplexParSpecIntVal);
witReturnCode witGetCplexParSpecName
(    WitRun * const theWitRun,
     char * *       cplexParSpecName);
     witBoolean *   cplexSelected);
witReturnCode witGetCplexStatusCode
(    WitRun * const theWitRun,
     int *          cplexStatusCode);
witReturnCode witGetCplexStatusText
(    WitRun * const theWitRun,
     char * *       cplexStatusText);
witReturnCode witGetCurrentObjective
(    WitRun * const theWitRun,
     char * * currentObjective );
witReturnCode witGetCurrentScenario
(    WitRun * const theWitRun,
     int * currentScenario );
witReturnCode witGetEquitability
(    WitRun * const theWitRun,
     int * equitability );
witReturnCode witGetExecEmptyBom
(    WitRun * const theWitRun,
     int * execEmptyBom );
witReturnCode witGetExpCutoff
(    WitRun * const theWitRun,
     float * expCutoff );
witReturnCode witGetExtOptActive
(    WitRun * const theWitRun,
     witBoolean * extOptActive );
witReturnCode witGetFeasible
(    WitRun * const theWitRun,
     witBoolean * feasible );
witReturnCode witGetForcedMultiEq
(    WitRun * const theWitRun,
    witBoolean * forcedMultiEq );
witReturnCode witGetHeurAllocActive
(   WitRun * const theWitRun,
    witBoolean * heurAllocActive );
witReturnCode witGetIndependentOffsets
(   WitRun * const theWitRun,
    witBoolean * independentOffsets );
witReturnCode witGetLotSizeTol
(   WitRun * const theWitRun,
    float * lotSizeTol );
witReturnCode witGetMinimalExcess
(   WitRun * const theWitRun,
    witBoolean * minimalExcess );
witReturnCode witGetMipMode
(   WitRun * const theWitRun,
    witBoolean * mipMode );
witReturnCode witGetModHeurAlloc
(   WitRun * const theWitRun,
    witBoolean * modHeurAlloc );
witReturnCode witGetMultiExec
(   WitRun * const theWitRun,
    witBoolean * multiExec );
witReturnCode witGetMultiObjMode
(   WitRun * const theWitRun,
    witBoolean * multiObjMode );
witReturnCode witGetMultiObjTol
(   WitRun * const theWitRun,
    float * multiObjTol );
witReturnCode witGetMultiRoute
(   WitRun * const theWitRun,
    witBoolean * multiRoute );
witReturnCode witGetNPeriods
(   WitRun * const theWitRun,
    int * nPeriods );
witReturnCode witGetNScenarios
(   WitRun * const theWitRun,
    int * nScenarios );
witReturnCode witGetNstnResidual
(   WitRun * const theWitRun,
    witBoolean * nstnResidual );
witReturnCode witGetObjValue
(   WitRun * const theWitRun,
    witBoolean * objValue );
witReturnCode witGetObjectiveList
```

```
(    WitRun * const theWitRun,
     int * lenList
     char * * * objectiveList );
witReturnCode witGetObjectiveListSpec
(    WitRun * const theWitRun,
     char * * objectiveListSpec );
witReturnCode witGetObjectiveSeqNo
(    WitRun * const theWitRun,
     int * objectiveSeqNo );
witReturnCode witGetOptInitMethod
(    WitRun * const theWitRun,
     witAttr * optInitMethod );
witReturnCode witGetOutputPrecision
(    WitRun * const theWitRun,
     int * outputPrecision );
witReturnCode witGetPenExec
(    WitRun * const theWitRun,
     witBoolean * penExec );
witReturnCode witGetPerfPegging
(    WitRun * const theWitRun,
     witBoolean * perfPegging );
witReturnCode witGetPeriodStage
(    WitRun * const theWitRun,
     int * * periodStage );
witReturnCode witGetPgdCritListMode
(    WitRun * const theWitRun,
     witBoolean * pgdCritListMode );
witReturnCode witGetPipExists
(    WitRun * const theWitRun,
     witBoolean * pipExists );
witReturnCode witGetPipSeqFromHeur
(    WitRun * const theWitRun,
     witBoolean * pipSeqFromHeur );
witReturnCode witGetPostprocessed
(    WitRun * const theWitRun,
     witBoolean * postprocessed );
witReturnCode witGetPrefHighStockSLBs
(    WitRun * const theWitRun,
     witBoolean * prefHighStockSLBs );
witReturnCode witGetPreprocessed
(    WitRun * const theWitRun,
     witBoolean * preprocessed );
witReturnCode witGetProbability
(    WitRun * const theWitRun,
     float * probability );
witReturnCode witGetRespectStockSLBs
(    WitRun * const theWitRun,
     witBoolean * respectStockSLBs );
witReturnCode witGetRoundReqVols
(    WitRun * const theWitRun,
     witBoolean * roundReqVols );
witReturnCode witGetSkipFailures
(    WitRun * const theWitRun,
     witBoolean * skipFailures );
witReturnCode witGetSelSplit
(    WitRun * const theWitRun,
     witBoolean * selSplit );
witReturnCode witGetSelectionRecovery
(    WitRun * const theWitRun,
     witBoolean * selectionRecovery );
witReturnCode witGetSolverLogFileName
(    WitRun * const theWitRun,
     char * * solverLogFileName );
witReturnCode witGetStageByObject
(    WitRun * const theWitRun,
     witBoolean * stageByObject );
witReturnCode witGetStochMode
(    WitRun * const theWitRun,
     witBoolean * stochMode );
witReturnCode witGetStochSolnMode
(    WitRun * const theWitRun,
     witBoolean * stochSolnMode );
witReturnCode witGetStockReallocation
(    WitRun * const theWitRun,
     witBoolean * stockReallocation );
witReturnCode witGetTieBreakPropRt
(    WitRun * const theWitRun,
     witBoolean * tieBreakPropRt );
witReturnCode witGetTitle
(    WitRun * const theWitRun,
     char * * title );
witReturnCode witGetTruncOffsets
(    WitRun * const theWitRun,
     witBoolean * truncOffsets );
witReturnCode witGetTwoWayMultiExec
(    WitRun * const theWitRun,
     witBoolean * twoWayMultiExec );
witReturnCode witGetUseFocusHorizons
```

```
(    WitRun * const theWitRun,
     witBoolean * useFocusHorizons );
witReturnCode witGetUserHeurStart
(    WitRun * const theWitRun,
     witBoolean * userHeurStart );
witReturnCode witGetWbounds
(    WitRun * const theWitRun,
     float * wbounds );
```

Description

`theWitRun`

Identifies the WIT problem to be used by this function.

`value`

Location where the attribute value is to be returned.

Usage Notes

1. When the type of the second parameter value is `char * * or int * *`, a vector is returned. It is the application's responsibility to free the returned vector.
2. Concerning `witGetObjectiveList` — On return, the `lenList` argument gives the length of the list of objective names.
3. Concerning `witGetObjectiveList` — It is the responsibility of the application to free the returned list of character arrays.

Error conditions

- Any violations of the requirements listed in "Global (WIT Problem) Attributes" on page 98.
- For stochastic implosion attributes, see Table 2 on page 27.

Example

```
float   wbounds;
int     nPeriods;
witBoolean accAfterOptImp;
witGetWbounds( theWitRun, &wbounds );
witGetNperiods( theWitRun, &nPeriods );
printf( "wbounds = %f\n", wbounds );
printf( "nPeriods = %d\n", nPeriods );
witGetaccAfterOptImp( theWitRun, &accAfterOptImp );
printf("accAfterOptImp is currently %s\n",
        accAfterOptImp ? "TRUE" : "FALSE );
```

witGetCriticalList

```
witReturnCode witGetCriticalList
(    WitRun * const theWitRun,
     int * lenCritList,
     char * * * critPartList,
     int * * critPeriod );
```

Description

This function returns the list of parts identified by `witHeurImplode` or `witOptImplode` as being critical in the indicated period. Critical parts have limited supply volume which is preventing implosion from finding a better solution.

`theWitRun`

Identifies the WIT problem to be used by this function.

`lenCritList`

On return this contains the number of critical parts and periods found by the implosion functions.

`critPartList`

On return this contains the list of critical part names ordered from most significant to least significant. The length of the returned vector is `lenCritList`.

`critPeriod`

On return this contains the period in which the corresponding part in critPartList had a limiting supply volume. The length of the returned vector is `lenCritList`.

Part `critPartList[i]` is critical in period `critPeriod[i]`,

For all i = 0, 1, …`lenCritList -` 1

Usage Notes

1. It is the responsibility of the application to free the returned vectors.
2. The Critical Parts List will only be computed by the implosion functions if `computeCriticalList` has been set to `WitTRUE.`
3. The Critical Parts List will be empty, if the WitRun is in an unpostprocessed state.
4. The returned value of `lenCritList` will be ≥ 0.
5. Must follow `witOptImplode` or `witHeurImplode,` else a critical list of length zero is returned.

Example

```
char ** critList;
int * critPer;
int listLen,i;
witGetCriticalList(theWitRun,
    &listLen, &critList, &critPer );
for ( i=0; i<listLen; i++ )
    printf( "Number %d critical part %s in period %d\n",
             i, critList[i], critPer[i] );
for ( i=0; i<listLen; i++ ) free( critList[i] );
free( critList );
free( critPer );
```

The Critical Parts List is obtained and displayed on `stdout`. The memory obtained by `witGetCriticalList` is freed.

witGetFocusShortageVol

```
witReturnCode witFocusShortageVol
(    WitRun * const theWitRun,
     int * lenList,
     char *** partList
     float *** focusShortageVolList );
```

Description

Returns list of parts with non-zero `focusShortageVol` and their `focusShortageVol`. For more details, see “Focussed Shortage Schedule” on page 40.

`theWitRun`

Identifies the WIT problem to be used by this function

`lenList`

On return contains the number of parts in `partList` and the number of shortage volume vectors in `focusShortageVolList`.

`partList`

On return contains a list of parts with nonzero `focusShortageVol`.

`focusShortageVolList`

On return contains a list of vectors. Each vector contains the `focusShortageVol` for the corresponding part in `partList`. The length of the vectors is `nPeriods`.

Usage Notes

1. This function causes WIT to compute the FSS if needed.
2. It is the application’s responsibility to free the returned vectors.

Error Conditions

- The WitRun must be in a postprocessed state. See “The State of a WitRun” on page 161.
- The current implosion solution must not be user-specified. See “User-Specified Solution” on page 37

Example

```
char ** partList;
float ** focusShortageVolList;
int lenList, nPeriods, i, t;
witGetNPeriods ( theWitRun,&nPeriods);
witGetFocusShortageList ( theWitRun,&lenList,&partList
             &focusShortageVolList);
for ( i=0; i<lenList; i++ ) {
  printf("Focussed Shortage Volume for part %s is:\n",
         partList[i]);
 for ( t=0; t<nPeriods; t++ )
  printf("   %f",focusShortageVolList[i][t]);
  printf("\n");
  free ( partList[i]);
  free ( focusShortageVolList[i]);
}
free ( partList);
free ( focusShortageVolList);
```

witGetOperations

```
witReturnCode witGetOperations
(    WitRun * const theWitRun,
     int * lenOperationList,
     char *** operationList );
```

Description

This function returns the list of operationNames of all operations.

`theWitRun`

Identifies the WIT problem to be used by this function.

`lenOperationList`

On return this contains the number of operations returned in operationList.

`operationList`

On return this contains the list of the operationNames of all operations defined in theWitRun.

Usage Notes

1. It is the responsibility of the application to free the returned vector.
2. If this function is called more than once, the order in which the operations appear may change from one call to the next.

Example

```
char ** operationList;
int listLen, i;
witGetOperations (theWitRun, &listLen, &operationList );
for ( i=0; i<listLen; i++ )
     printf( "Operation name: %s\n",operationList[i] );
for ( i=0; i<listLen; i++ ) free( operationList[i] );
free( operationList );
```

The list of all operation names is obtained and printed to `stdout.` The memory obtained by `witGetOperations` is freed.

witGetParts

```
witReturnCode witGetParts
(    WitRun * const theWitRun,
     int * lenPartList,
     char * * * partList);
```

Description

This function returns the list of partNames of all parts.

`theWitRun`

Identifies the WIT problem to be used by this function.

`lenPartList`

On return this contains the number of parts returned in partList.

`partList`

On return this contains the list of the partNames of all parts defined in theWitRun.

Usage Notes

1. It is the responsibility of the application to free the returned vector.
2. If this function is called more than once, the order in which the operations appear may change from one call to the next.

Example

```
char ** partList;
int listLen, i;
witGetParts (theWitRun, &listLen, &partList );
for ( i=0; i<listLen; i++ )
     printf( "Part name: %s\n",partList[i] );
for ( i=0; i<listLen; i++ ) free( partList[i] );
free( partList );
```

The list of all part names is obtained and printed to `stdout.` The memory obtained by `witGetParts` is freed.

witGetPgdCritList

```
witReturnCode witGetPgdCritList
(    WitRun * const theWitRun,
     int *          lenLists,
     char * * *     critPartNameList,
     int * *        critPerList,
     char * * *     demPartNameList,
     char * * *     demandNameList,
     int * *        shipPerList);
```

Description

This function retrieves the pegged critical list. See “Pegged Criticial List” on page 78.

`theWitRun`

Identifies the WIT problem to be used by this function.

`lenLists`

On return `(* lenLists)` is the number of elements in the pegged critical list.

`critPartNameList`

On return: for `0 ≤ i < (* lenLists)`:

`(* critPartNameList)[i]` is the partName of the critical part for element #i in the pegged critical list.

`critPerList`

On return: for `0 ≤ i < (* lenLists): (* critPerList)[i]`

is the critical period for element #i in the pegged critical list.

`demPartNameList`

On return: for `0 ≤ i < (* lenLists)`:

`(* demPartNameList)[i]` is the demandedPartName of the demand for element #i in the pegged critical list.

`demandNameList`

On return: for `0 ≤ i < (* lenLists)`:

`(* demandNameList)[i]` is the demandName of the demand for element #i in the pegged critical list.

`shipPerList`

On return: for `0 ≤ i < (* lenLists): (* shipPerList)[i]`

is the shipment period for element #i in the pegged critical list.

Usage Notes

1. It is the responsibility of the application to free the returned vectors.

2. The pegged critical list will be non-empty only if heuristic implosion or heuristic allocation has been invoked while `pgdCritListMode` was `WitTRUE and pgdCritListMode` has not been set to `WitFALSE` since then.

Example

```
void printPgdCritList (WitRun * theWitRun)
   {
   int      lenLists;
   char * * critPartNameList;
   int *    critPerList;
   char * * demPartNameList;
   char * * demandNameList;
   int *    shipPerList;
   int      theIdx;

   witGetPgdCritList (
        theWitRun,
      & lenLists,
      & critPartNameList,
      & critPerList,
      & demPartNameList,
      & demandNameList,
      & shipPerList);

   printf (
      "Idx  Crit  Crit  Dem    Dem   Ship\n"
      "     Part   Per  Part          Per\n");
for (theIdx = 0; theIdx < lenLists; theIdx ++)
   printf (
      "%3d  %-4s  %4d  %-5s  %-4s  %4d\n",
      theIdx,
      critPartNameList[theIdx],
      critPerList     [theIdx],
      demPartNameList [theIdx],
      demandNameList  [theIdx],
      shipPerList     [theIdx]);

for (theIdx = 0; theIdx < lenLists; theIdx ++)
   {
   free (critPartNameList[theIdx]);
   free (demPartNameList [theIdx]);
   free (demandNameList  [theIdx]);
   }

free (critPartNameList);
free (critPerList);
free (demPartNameList);
free (demandNameList);
free (shipPerList);
}
```

The output of the function might be as follows:

```
Idx  Crit  Crit  Dem    Dem   Ship
     Part   Per  Part          Per
  0  CapA     2  ProdC  DemE     3
  1  CapA     2  ProdD  DemF     5
  2  CapB     1  ProdD  DemF     5
  3  CapA     2  ProdD  DemF     5
```

The interpretation of this output would be as follows:

- First a shortage in the supplyVol of part CapA in period 2 prevented a potential increase to the shipVol of demand DemE for part ProdC in period 3.
- Next a shortage in the supplyVol of part CapA in period 2 prevented a potential increase to the shipVol of demand DemF for part ProdD in period 5.
- Next a shortage in the supplyVol of part CapB in period 1 prevented a potential increase to the shipVol of demand DemF for part ProdD in period 5.
- Finally a shortage in the supplyVol of part CapA in period 2 prevented a potential increase to the shipVol of demand DemF for part ProdD in period 5 (again).

witSetAttribute

```
witReturnCode witSetAttribute
(   WitRun * const theWitRun,
    Type value );
```

`witSetAttribute` represents a group of functions for setting WIT global attributes. For more information on global attributes see “Global (WIT Problem) Attributes” on page 98.

The `witSetAttribute` functions are:

```
witReturnCode witSetAccAfterOptImp
(   WitRun * const theWitRun,
    const witBoolean accAfterOptImp );
witReturnCode witSetAccAfterSoftLB
(   WitRun * const theWitRun,
    const witBoolean accAfterSoftLB );
witReturnCode witSetAppData
(   WitRun * const theWitRun,
    void * const appData );
witReturnCode witSetAutoPriority
(   WitRun * const theWitRun,
    const witBoolean autoPriority );
witReturnCode witSetCompPrices
(   WitRun * const theWitRun,
    const witBoolean compPrices );
witReturnCode witSetComputeCriticalList
(   WitRun * const theWitRun,
    const witBoolean computeCriticalList );
witReturnCode witSetCplexParSpecDblVal
(   WitRun * const theWitRun,
    const float cplexParSpecDblVal );
witReturnCode witSetCplexParSpecIntVal
(   WitRun * const theWitRun,
    const int cplexParSpecIntVal );
witReturnCode witSetCplexParSpecName
(   WitRun * const theWitRun,
    const char * const cplexParSpecName );
witReturnCode witSetCurrentObjective
(   WitRun * const theWitRun,
    const char * const currentObjective );
witReturnCode witSetEquitability
(   WitRun * const theWitRun,
    const int equitability );
witReturnCode witSetExecEmptyBom
```

```
(    WitRun * const theWitRun,
     const witBoolean execEmptyBom );
witReturnCode witSetExpCutoff
(    WitRun * const theWitRun,
     const float expCutoff );
witReturnCode witSetIndependentOffsets
(    WitRun * const theWitRun,
     const witBoolean independentOffsets );
witReturnCode witSetLotSizeTol
(    WitRun * const theWitRun,
     const float lotSizeTol );
witReturnCode witSetMinimalExcess
(    WitRun * const theWitRun,
     const witBoolean minimalExcess );
witReturnCode witSetMipMode
(    WitRun * const theWitRun,
     const witBoolean mipMode );
witReturnCode witSetModHeurAlloc
(    WitRun * const theWitRun,
     const witBoolean modHeurAlloc );
witReturnCode witSetMultiExec
(    WitRun * const theWitRun,
     const witBoolean multiExec );
witReturnCode witSetMultiObjMode
(    WitRun * const theWitRun,
     const witBoolean multiObjMode );
witReturnCode witSetMultiObjTol
(    WitRun * const theWitRun,
     const float multiObjTol );
witReturnCode witSetMultiRoute
(    WitRun * const theWitRun,
     const witBoolean multiRoute );
witReturnCode witSetNPeriods
(    WitRun * const theWitRun,
     const int nPeriods );
witReturnCode witSetNScenarios
(    WitRun * const theWitRun,
     const int nScenarios );
witReturnCode witSetNstnResidual
(    WitRun * const theWitRun,
     const witBoolean nstnResidual );
witReturnCode witSetObjectiveList
(    WitRun * const theWitRun,
     const int lenList
```

```
    const char * const * const objectiveList );
witReturnCode witSetObjectiveListSpec
(   WitRun * const theWitRun,
    const char * const objectiveListSpec );
witReturnCode witSetObjectiveSeqNo
(   WitRun * const theWitRun,
    const int objectiveSeqNo );
witReturnCode witSetOptInitMethod
(   WitRun * const theWitRun,
    const witAttr optInitMethod );
witReturnCode witSetOutputPrecision
(   WitRun * const theWitRun,
    int outputPrecision );
witReturnCode witSetPenExec
(   WitRun * const theWitRun,
    const witBoolean penExec );
witReturnCode witSetPerfPegging
(   WitRun * const theWitRun,
    const witBoolean perfPegging );
witReturnCode witSetPeriodStage
(   WitRun * const theWitRun,
    const int * periodStage );
witReturnCode witSetPgdCritListMode
(   WitRun * const theWitRun,
    const witBoolean pgdCritListMode );
witReturnCode witSetPipSeqFromHeur
(   WitRun * const theWitRun,
    const witBoolean pipSeqFromHeur );
witReturnCode witSetPrefHighStockSLBs
(   WitRun * const theWitRun,
    const witBoolean prefHighStockSLBs );
witReturnCode witSetProbability
(   WitRun * const theWitRun,
    const float probability );
witReturnCode witSetRespectStockSLBs
(   WitRun * const theWitRun,
    const witBoolean respectStockSLBs );
witReturnCode witSetRoundReqVols
(   WitRun * const theWitRun,
    const witBoolean roundReqVols );
witReturnCode witSetSkipFailures
(   WitRun * const theWitRun,
    const witBoolean skipFailures );
witReturnCode witSetSelSplit
```

```
(    WitRun * const theWitRun,
     const witBoolean selSplit );
witReturnCode witSetSelectionRecovery
(    WitRun * const theWitRun,
     const witBoolean selectionRecovery );
witReturnCode witSetSolverLogFileName
(    WitRun * const theWitRun,
     const char * const solverLogFileName );
witReturnCode witSetStageByObject
(    WitRun * const theWitRun,
     const witBoolean stageByObject );
witReturnCode witSetStochMode
(    WitRun * const theWitRun,
     const witBoolean stochMode );
witReturnCode witSetStockReallocation
(    WitRun * const theWitRun,
     const witBoolean stockReallocation );
witReturnCode witSetTieBreakPropRt
(    WitRun * const theWitRun,
     const witBoolean tieBreakPropRt );
witReturnCode witSetTitle
(    WitRun * const theWitRun,
     const char * const title );
witReturnCode witSetTruncOffsets
(    WitRun * const theWitRun,
     const witBoolean truncOffsets );
witReturnCode witSetTwoWayMultiExec
(    WitRun * const theWitRun,
     const witBoolean twoWayMultiExec );
witReturnCode witSetUseFocusHorizons
(    WitRun * const theWitRun,
     const witBoolean useFocusHorizons );
witReturnCode witSetUserHeurStart
(    WitRun * const theWitRun,
     const witBoolean userHeurStart );
witReturnCode witSetWbounds
(    WitRun * const theWitRun,
     const float wbounds );
```

Description

`theWitRun`

Identifies the WIT problem to be used by this function.

`value`

The new value of the global attribute.

Usage notes

1. Setting some attributes will cause the state to change.   See “The State of a WitRun” on page 161.
2. Concerning `witSetObjectiveList` — The `lenList` argument specifies the length of the list of objective names.
3. Concerning `witSetOptInitMethod`—The valid values of the input parameter `optInitMethod` are:
    - `WitHEUR_OPT_INIT_METHOD`, which sets the optInitMethod attribute to “heuristic”.
    - `WitACC_OPT_INIT_METHOD`, which sets the optInitMethod attribute to “accelerated”.
    - `WitSCHED_OPT_INIT_METHOD`, which sets the optInitMethod attribute to “schedule”.
    - `WitCRASH_OPT_INIT_METHOD`, which sets the optInitMethod to "crash".

Error conditions

- Any violations of the requirements listed in “Global (WIT Problem) Attributes” on page 98.
- For stochastic implosion attributes, see Table 2 on page 27.
- Changing the value of the stochMode attribute from FALSE to TRUE causes WIT to enter stochastic mode. See the note on page 28 for a list of conditions checked when WIT to enters stochastic mode.

Exceptions to General Error Conditions

- In `witSetAppData`, the `appData` argument is allowed to be a NULL pointer.

Example

```
witReturnCode rc;
rc = witSetAutoPriority(theWitRun,WitTRUE);
rc = witSetTitle(theWitRun,"Plant 3, Spring 93");
```

witSetCurrentScenario

```
witReturnCode witSetCurrentScenario (
   WitRun * const theWitRun,
   const int      currentScenario);
```

Description

This function sets the value of the currentScenario attribute. See “Stochastic Implosion” on page 21.

`theWitRun`

Identifies the WIT problem to be used by this function.

`currentScenario`

The new value to be assigned to the currentScenario attribute.

Error conditions

- stochMode must be TRUE.

Example

See “Sample 5: newsVendor.C” on page 433.

CPU-time efficiency

There are two important things to know about the CPU speed of `witSetCurrentScenario`:

- `witSetCurrentScenario` expends significant CPU time.
- `witSetCurrentScenario` expends much more CPU time when stochSolnMode = TRUE than when stochSolnMode = FALSE.

As a consequence, the order in which scenario-specific attributes are entered or retrieved can potentially affect the run-time speed of an application program. While there are many ways to organize the setting and retrieving of these attributes, for clarity, two contrasting approaches will be discussed here: the “scenario-major” approach and the “object-major” approach:

- In the scenario-major approach, the calls to the functions that set or retrieve scenario-specific attributes are organized in a two-level loop: The outer loop iterates through all scenarios, while the inner loop iterates through all pertinent WIT data objects (e.g., parts and demands, etc.).
- In the object-major approach, the calls are also organized in a two-level loop: In this case, the outer loop iterates through all pertinent WIT data objects, while the inner loop iterates through all scenarios.

For example, retrieving the shipVol attribute using the scenario-major approach might look like this:

```
for (theIdx = 0; theIdx < nScenarios; theIdx ++)
   {
   witSetCurrentScenario (theWitRun, theIdx);

   for (...) // Loop through all demands.
      {
      witGetDemandShipVol (...);
      }
   }
```

Retrieving the shipVol attribute using the object-major approach might look like this:

```
for (...) // Loop through all demands.
   {
   for (theIdx = 0; theIdx < nScenarios; theIdx ++)
      {
      witSetCurrentScenario (theWitRun, theIdx);

      witGetDemandShipVol (...);
      }
   }
```

On large problems, it’s clear that the object-major approach would make many more calls to `witSetCurrentScenario` than the scenario-major approach would. In light of this, the following general tendencies can be expected:

- When entering scenario-specific data, stochSolnMode will be FALSE. In this case, the object-major approach will tend to be quite a bit slower than the scenario-major approach.
- When extracting scenario-specific data, stochSolnMode will be TRUE. In this case, the object-major approach will tend to be vastly slower than the scenario-major approach.

.

Part Functions

The following functions allow the application program to access and modify data associated with a specific part.

witAddPart

```
witReturnCode witAddPart
(    WitRun * const theWitRun,
     const char * const partName,
     const witAttr partCategory );
```

Description

This function is used to create a part with default attribute values.

`theWitRun`

Identifies the WIT problem to be used by this function.

`partName`

The partName for the new part.

`partCategory`

The partCategory for the new part. The allowed choices are `WitMATERIAL` and `WitCAPACITY`.

Usage notes

1. The default values for part attribute data are defined in “Part Attributes” on page 118.

Error conditions

- Any violations of the requirements listed in “Part Attributes” on page 118.
- The object iteration process must not be active.

Example

```
witReturnCode rc;
rc = witAddPart(theWitRun,"prod1",WitMATERIAL);
rc = witAddPart(theWitRun,"raw1",WitMATERIAL);
rc = witAddPart(theWitRun,"raw2",WitMATERIAL);
rc = witAddPart(theWitRun,"machine1",WitCAPACITY);
```

Four parts are created and added to the WIT data structures.

witAddPartWithOperation

```
witReturnCode witAddPartWithOperation
(    WitRun * const theWitRun,
     const char * const partAndOperationName );
```

Description

This function adds a material part, an operation and a connecting BOP entry. For more information, see “Part-With-Operation” on page 10.

`theWitRun`

Identifies the WIT problem to be used by this function.

`partAndOperationName`

The partName for the new part and the operationName for the new operation.

Usage notes

1. The default values for part attribute data, operation attribute data, and BOP entry attribute data are defined in “Part Attributes” on page 118, “Operation Attributes” on page 131, and “BOP Entry Attributes” on page 148.

Error conditions

- Any violations of the requirements listed in “Part Attributes” on page 118 and “Operation Attributes” on page 131.
- The object iteration process must not be active.

Example

```
witReturnCode rc;
rc = witAddPartWithOperation(theWitRun,"Part 1");
```

A material part named "Part 1" is added, an operation named "Part 1" is added, and a BOP entry is added to connect part "Part 1" with operation "Part 1".

witGetPartAttribute

```
witReturnCode witGetPartAttribute
(    WitRun * const theWitRun,
     const char * const partName,
     Type value );
```

`witGetPartAttribute` represents a group of functions for obtaining the value of part attributes. For more information on part attributes see “Part Attributes” on page 118.

The `witGetPartAttribute` functions are:

```
witReturnCode witGetPartAppData
(    WitRun * const theWitRun,
     const char * const partName,
     void ** appData );
witReturnCode witGetPartAsapPipOrder
(    WitRun * const theWitRun,
     const char * const partName,
     witBoolean * asapPipOrder );
witReturnCode witGetPartBoundedLeadTimes
(    WitRun * const theWitRun,
     const char * const partName,
     witBoolean * boundedLeadTimes );
witReturnCode witGetPartBuildAheadUB
(    WitRun * const theWitRun,
     const char * const partName,
     int ** buildAheadUB );
witReturnCode witGetPartBuildAsap
(    WitRun * const theWitRun,
     const char * const partName,
     witBoolean * buildAsap );
witReturnCode witGetPartBuildNstn
(    WitRun * const theWitRun,
     const char * const partName,
     witBoolean * buildNstn );
witReturnCode witGetPartCategory
(    WitRun * const theWitRun,
     const char * const partName,
     witAttr * partCategory );
witReturnCode witGetPartConsVol
(    WitRun * const theWitRun,
     const char * const partName,
     float ** consVol );
witReturnCode witGetPartExcessVol
```

```
(    WitRun * const theWitRun,
     const char * const partName,
     float ** excessVol );
witReturnCode witGetPartFocusShortageVol
(    WitRun * const theWitRun,
     const char * const partName,
     float ** focusShortageVol );
witReturnCode witGetPartMrpConsVol
(    WitRun * const theWitRun,
     const char * const partName,
     float ** mrpConsVol );
witReturnCode witGetPartMrpExcessVol
(    WitRun * const theWitRun,
     const char * const partName,
     float ** mrpExcessVol );
witReturnCode witGetPartMrpResidualVol
(    WitRun * const theWitRun,
     const char * const partName,
     float ** mrpResidualVol );
witReturnCode witGetPartObjectStage
(    WitRun * const theWitRun,
     const char * const partName,
     int * objectStage );
witReturnCode witGetPartProdVol
(    WitRun * const theWitRun,
     const char * const partName,
     float ** prodVol );
witReturnCode witGetPartPropRtg
(    WitRun * const theWitRun,
     const char * const partName,
     witBoolean ** propRtg );
witReturnCode witGetPartReqVol
(    WitRun * const theWitRun,
     const char * const partName,
     float ** reqVol );
witReturnCode witGetPartResidualVol
(    WitRun * const theWitRun,
     const char * const partName,
     float ** residualVol );
witReturnCode witGetPartScrapAllowed
(    WitRun * const theWitRun,
     const char * const partName,
     witBoolean * scrapAllowed );
witReturnCode witGetPartScrapCost
```

```
(    WitRun * const theWitRun,
     const char * const partName,
     float ** scrapCost );
witReturnCode witGetPartScrapVol
(    WitRun * const theWitRun,
     const char * const partName,
     float ** scrapVol );
witReturnCode witGetPartSelForDel
(    WitRun * const theWitRun,
     const char * const partName,
     witBoolean * selForDel );
witReturnCode witGetPartShadowPrice
(    WitRun * const theWitRun,
     const char * const partName,
     float ** shadowPrice );
witReturnCode witGetPartSingleSource
(    WitRun * const theWitRun,
     const char * const partName,
     witBoolean * singleSource );
witReturnCode witGetPartStockBounds
(    WitRun * const theWitRun,
     const char * const partName,
     float ** hardLower ),
     float ** softLower ),
     float ** hardUpper );
witReturnCode witGetPartStockCost
(    WitRun * const theWitRun,
     const char * const partName,
     float ** stockCost );
witReturnCode witGetPartStockVol
(    WitRun * const theWitRun,
     const char * const partName,
     float ** stockVol );
witReturnCode witGetPartSupplyVol
(    WitRun * const theWitRun,
     const char * const partName,
     float ** supplyVol );
```

Description

`theWitRun`

Identifies the WIT problem to be used by this function.

`partName`

The partName of the part whose attribute value is to be returned.

`value`

The location where the attribute value is returned.

Usage notes

1. When the type of the third parameter value is `float **`, then a `float *` vector with length `nPeriods` is returned. It is the responsibility of the application to free the returned vector.
2. Concerning `witGetPartConsVol`, `witGetPartExcessVol`, `witGetPartScrapVol`, `witGetPartStockVol`, `witGetPartProdVol`—meaningful values will be returned only if `theWitRun` is in a postprocessed state. See “The State of a WitRun” on page 161.
3. Concerning `witGetPartMrpConsVol`, `witGetPartMrpExcessVol`, `witGetPartReqVol` —meaningful values will be returned only if a WIT-MRP solution exists.
4. Concerning `witGetPartFocusShortageVol` — this function causes WIT to compute the FSS if needed.

Error conditions

- A part with the specified `partName` has not been previously defined.
- Any violations of the requirements listed in “Part Attributes” on page 118.
- Concerning `witGetPartFocusShortageVol`— The WitRun must be in a postprocessed state. See “The State of a WitRun” on page 161.
- Also concerning `witGetPartFocusShortageVol`— The current implosion solution must not be user-specified. See “User-Specified Solution” on page 37.
- For stochastic implosion attributes, see Table 2 on page 27.

Example

```
float * prodVol;
float * stockVol;
int nPeriods, t;
witAttr partCategory;
witGetPartProdVol ( theWitRun, "prod1", &prodVol);
witGetPartStockVol ( theWitRun, "prod1", &stockVol);
witGetNperiods ( theWitRun, &nPeriods);
for (t=0; t <nPeriods; t++){
     printf( "stockVol[%d] = %f\n", stockVol[t]);
     printf( "prodVol[%d]= %f\n", prodVol[t]);
}
free (stockVol);
free (prodVol);
witGetPartCategory(theWitRun, "partAbc", &partCategory );
     if ( partCategory == WitMATERIAL )
```

```
        printf( "partAbc is a material" );
    else if ( partCategory == WitCAPACITY )
        printf( "partAbc is a capacity" );
```

The per period production and stock volume for "prod1" is obtained and printed. The part category of "partAbc" is obtained and printed.

witGetPartBelowList

```
witReturnCode witGetPartBelowList
(    WitRun * const theWitRun,
     const char * const partName,
     int * lenList,
     char *** partNameList );
```

Description

This function returns the belowList attribute for the given part.

`theWitRun`

Identifies the WIT problem to be used by this function.

`partName`

The partName of the part whose below list is to be returned.

`lenList`

On return this contains the number of parts in the below list.

`partNameList`

On return this contains the list of partNames of the parts in the below list.

Usage Notes

1. It is the responsibility of the application to free the returned 2-dimensional array, `(* partNameList)`.
2. The below list is computed by WIT when it performs preprocessing of the data. If preprocessing has not already been performed when `witGetPartBelowList` is called, WIT will perform preprocessing automatically. See note 3 on page 238 for the performance implications of this situation.

Error Conditions

- `partName` does not identify an existing part.

Example

```
int      lenList;
char * * partNameList;
int      i;

witGetPartBelowList (
   theWitRun,
   "prod1",
   & lenList,
   & partNameList);

printf ("Below List for part prod1:\n");

for (i = 0; i < lenList; ++ i)
   printf ("   %s\n", partNameList[i]);

for (i = 0; i < lenList; ++ i)
   free (partNameList[i]);

free (partNameList);
```

The partNames of the below list for part “prod1” are obtained and printed to `stdout`. The memory allocated by `witGetPartBelowList` is freed.

witGetPartConsumingBomEntry

```
witReturnCode witGetPartConsumingBomEntry
(    WitRun * const theWitRun,
     const char * const partName,
     const int consIndex,
     char ** consumingOperationName,
     int * bomEntryIndex );
```

Description

A “consuming BOM entry” for a part is a BOM entry that indicates consumption of that part. This function returns the identifying attributes (consumingOperationName and bomEntryIndex) for a specified consuming BOM entry of a specified part.

`theWitRun`

Identifies the WIT problem to be used by this function.

`partName`

The partName of the specified part, i.e, the consumed part for the BOM entry whose identifying attributes are to be returned.

`consIndex`

An index that specifies which consuming BOM entry for the specified part will have its identifying attributes returned. `consIndex` is what distinguishes the specified consuming BOM entry from all other consuming BOM entries for the specified part.

`consumingOperationName`

On return contains the consumingOperationName of the specified consuming BOM entry.

`bomEntryIndex`

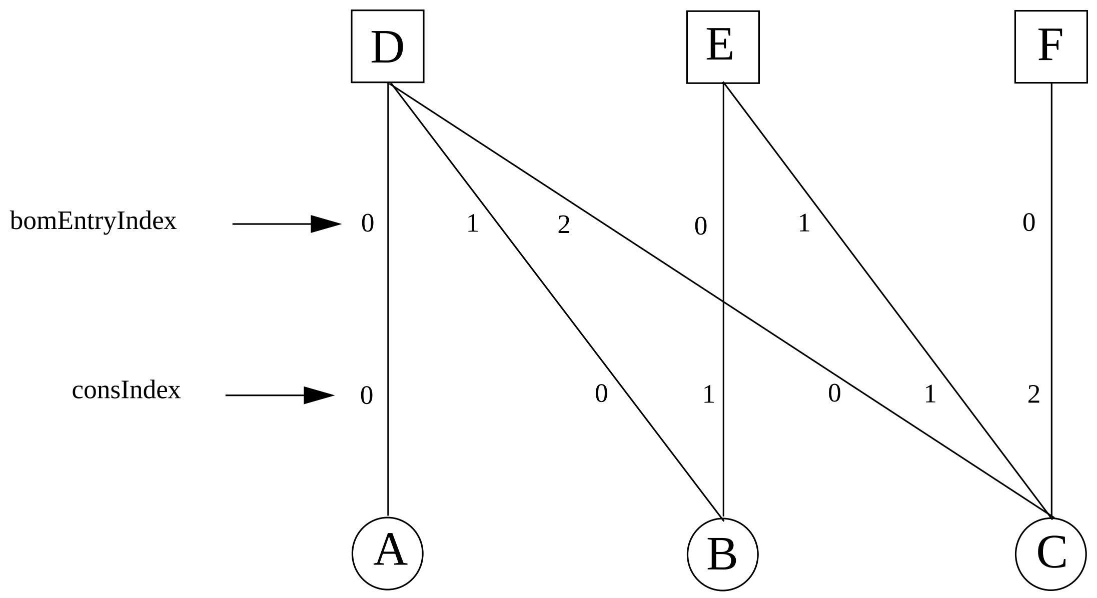
On return contains the bomEntryIndex of the specified consuming BOM entry.

Error Conditions

- `partName` does not match the partName of an existing part.
- A `consIndex` outside the range:

  $0 \leq$ `consIndex` < number of consuming BOM entries for the specified part. See “witGetPartNConsumingBomEntries” on page 209.

FIGURE 19 Example use of witGetPartNConsumingBomEntries and witGetPartConsumingBomEntry



Example use of witGetPartNConsumingBomEntries and witGetPartConsumingBomEntry

```
int    nConsumingBomEntries;
int    consIndex;
char * consumingOperationName;
int    bomEntryIndex;

witGetPartNConsumingBomEntries (
   theWitRun, "C", & nConsumingBomEntries);

printf (
  "Part C is consumed through the following %d "
  "BOM entries:\n\n",
  nConsumingBomEntries);

for (consIndex = 0;
     consIndex < nConsumingBomEntries;
     ++ consIndex)
   {
   witGetPartConsumingBomEntry (
      theWitRun,
      "C",
         consIndex,
         & consumingOperationName,
         & bomEntryIndex);

      printf (
        "   Operation %s, BOM entry #%d\n",
        consumingOperationName,
        bomEntryIndex);

      free (consumingOperationName);
      }
```

Given the problem shown above, this code generates the following output:

```
Part C is consumed through the following 3 BOM entries:

    Operation D, BOM entry #2
    Operation E, BOM entry #1
    Operation F, BOM entry #0
```

witGetPartConsumingSubsBomEntry

```
witReturnCode witGetPartConsumingSubsBomEntry
(    WitRun * const theWitRun,
     const char * const partName,
     const int consIndex,
     char ** consumingOperationName,
     int * bomEntryIndex,
     int * subsBomEntryIndex);
```

Description

A “consuming substitute BOM entry” for a part is a substitute BOM entry that indicates consumption of that part in place of another part. This function returns the identifying attributes (consumingOperationName, bomEntryIndex, and subsBomEntryIndex) for a specified consuming substitute BOM entry of a specified part.

`theWitRun`

Identifies the WIT problem to be used by this function.

`partName`

The partName of the specified part, i.e, the consumed part for the substitute BOM entry whose identifying attributes are to be returned.

`consIndex`

An index that specifies which consuming substitute BOM entry for the specified part will have its identifying attributes returned. `consIndex` is what distinguishes the specified consuming substitute BOM entry from all other consuming substitute BOM entries for the specified part.

`consumingOperationName`

On return contains the consumingOperationName of the specified consuming substitute BOM entry.

`bomEntryIndex`

On return contains the bomEntryIndex of the specified consuming substitute BOM entry.

`subsBomEntryIndex`

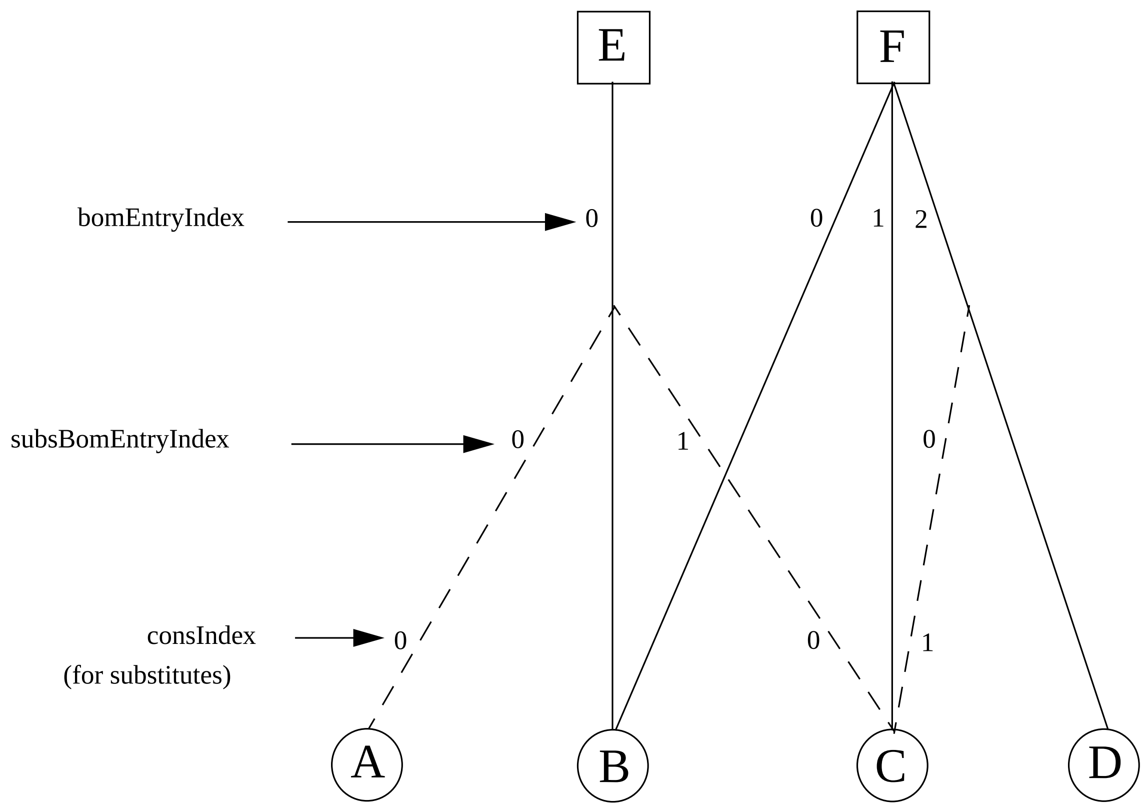
On return contains the subsBomEntryIndex of the specified consuming substitute BOM entry.

Error Conditions

- `partName` does not match the partName of an existing part.
- A `consIndex` outside the range:

  $0 \leq$ `consIndex` < number of consuming substitute BOM entries for the specified part. See “witGetSubsBomEntryAttribute” on page 260.

FIGURE 20 Example use of witGetPartNConsumingSubsBomEntries and witGetPartConsumingSubsBomEntry



Example use of witGetPartNConsumingSubsBomEntries and witGetPartConsumingSubsBomEntry.

```
int    nConsumingSubsBomEntries;
int    consIndex;
char * consumingOperationName;
int    bomEntryIndex;
int    subsBomEntryIndex;

witGetPartNConsumingSubsBomEntries (
   theWitRun, "C", & nConsumingSubsBomEntries);

printf (
  "Part C is consumed through the following %d\n"
  "substitute BOM entries:\n\n",
  nConsumingSubsBomEntries);

for (consIndex = 0;
    consIndex < nConsumingSubsBomEntries;
    ++ consIndex)
  {
  witGetPartConsumingSubsBomEntry (
     theWitRun,
     "C",
     consIndex,
     & consumingOperationName,
     & bomEntryIndex
     & subsBomEntryIndex);

  printf (
    "   Operation %s, BOM entry #%d, "
       "substitute BOM entry $%d\n",
    consumingOperationName,
    bomEntryIndex,
    subsBomEntryIndex);

  free (consumingOperationName);
  }
```

Given the problem shown above, this code generates the following output:

```
Part C is consumed through the following 2
substitute BOM entries:

    Operation E, BOM entry #0, substitute BOM entry #1
    Operation F, BOM entry #2, substitute BOM entry #0
```

witGetPartDemands

```
witReturnCode witGetPartDemands
(    WitRun * const theWitRun,
     const char * const demandedPartName,
     int * lenDemandList,
     char *** demandList);
```

Description

This function returns the list of the demandNames of the demands for the given part.

`theWitRun`

Identifies the WIT problem to be used by this function.

`demandedPartName`

The partName of the part whose demands are to be listed.

`lenDemandList`

On return, this contains the number of demandNames returned in `demandList`.

`demandList`

On return this contains the list of demandNames of the demands for the part indicated by `demandedPartName`.

Usage Notes

1. It is the responsibility of the application to free the returned vector.

Error Conditions

- `demandedPartName` does not identify an existing part.

Example

```
char ** demandList;
int listLen, i;
witGetPartDemands ( theWitRun,
              "prod1", &listLen, &demandList );
for ( i=0; i<listLen; i++ )
     printf( "Demand name: %s\n",demandList[i] );
for ( i=0; i<listLen; i++ ) free( demandList[i] );
free( demandList );
```

The demandNames of the demands for the part named "prod1" are obtained and printed to `stdout`. The memory obtained by `witGetPartDemands` is freed.

witGetPartExists

```
witReturnCode witGetPartExists
(    WitRun * const theWitRun,
     const char * const partName,
     witBoolean * exists);
```

Description

This function allows the application to determine if a specified part is defined in the WIT data structures.

`theWitRun`

Identifies the WIT problem to be used by this function.

`partName`

The partName of the part to be tested to see if it has already been defined.

`exists`

Returns `WitTRUE` if a part with the specified partName has been defined, otherwise `WitFALSE`.

Example

```
witBoolean exists;
witGetPartExists(theWitRun, "partA", &exists );
if (exists)
     printf ("partA has been defined.\n");
else
     printf ("partA has not been defined.\n");
```

witGetPartNConsumingBomEntries

```
witReturnCode witGetPartNConsumingBomEntries
(    WitRun * const theWitRun,
     const char * const partName,
     int * nConsumingBomEntries );
```

Description

This function returns the number of BOM entries that indicate consumption of a specified part.

`theWitRun`

Identifies the WIT problem to be used by this function.

`partName`

The partName of the specified part, i.e, the consumed part for all of the BOM entries being counted.

`nConsumingBomEntries`

On return this contains the number of BOM entries that indicate consumption of the specified part.

Error Conditions

- `partName` does not match the partName of an existing part.

Example

See "Example use of witGetPartNConsumingBomEntries and witGetPartConsumingBomEntry" on page 201.

witGetPartNConsumingSubsBomEntries

```
witReturnCode witGetPartNConsumingSubsBomEntries
(    WitRun * const theWitRun,
     const char * const partName,
     int * nConsumingSubsBomEntries );
```

Description

This function returns the number of substitute BOM entries that indicate consumption of the specified part in place of another part.

`theWitRun`

Identifies the WIT problem to be used by this function.

`partName`

The partName of the specified part, i.e, the consumed part for all of the substitute BOM entries being counted.

`nConsumingSubsBomEntries`

On return this contains the number of substitute BOM entries that indicate consumption of the specified part in place of another part.

Error Conditions

- `partName` does not match the partName of an existing part.

Example

See “Example use of witGetPartNConsumingSubsBomEntries and witGetPartConsumingSubsBomEntry.” on page 204.

witGetPartNProducingBopEntries

```
witReturnCode witGetPartNProducingBopEntries
(    WitRun * const theWitRun,
     const char * const partName,
     int * nProducingBopEntries );
```

Description

This function returns the number of BOP entries that indicate production of the specified part.

`theWitRun`

Identifies the WIT problem to be used by this function.

`partName`

The partName of the specified part, i.e, the produced part for all of the BOP entries being counted.

`nProducingBopEntries`

On return this contains the number of BOP entries that indicate production of the specified part.

Error Conditions

- `partName` does not match the partName of an existing part.

Example

See "Example use of witGetPartNProducingBopEntries and witGetPartProducingBopEntry." on page 213.

witGetPartProducingBopEntry

```
witReturnCode witGetPartProducingBopEntry
(    WitRun * const theWitRun,
     const char * const partName,
     const int prodIndex,
     char ** producingOperationName,
     int * bopEntryIndex );
```

Description

A “producing BOP entry” for a part is a BOP entry that indicates production of that part. This function returns the identifying attributes (producingOperationName and bopEntryIndex) for a specified producing BOP entry of a specified part.

`theWitRun`

Identifies the WIT problem to be used by this function.

`partName`

The partName of the specified part, i.e, the produced part for the BOP entry whose identifying attributes are to be returned.

`prodIndex`

An index that specifies which producing BOP entry for the specified part will have its identifying attributes returned. `prodIndex` is what distinguishes the specified producing BOP entry from all other producing BOP entries for the specified part.

`producingOperationName`

On return contains the name of the producing operation for the specified producing BOP entry.

`bopEntryIndex`

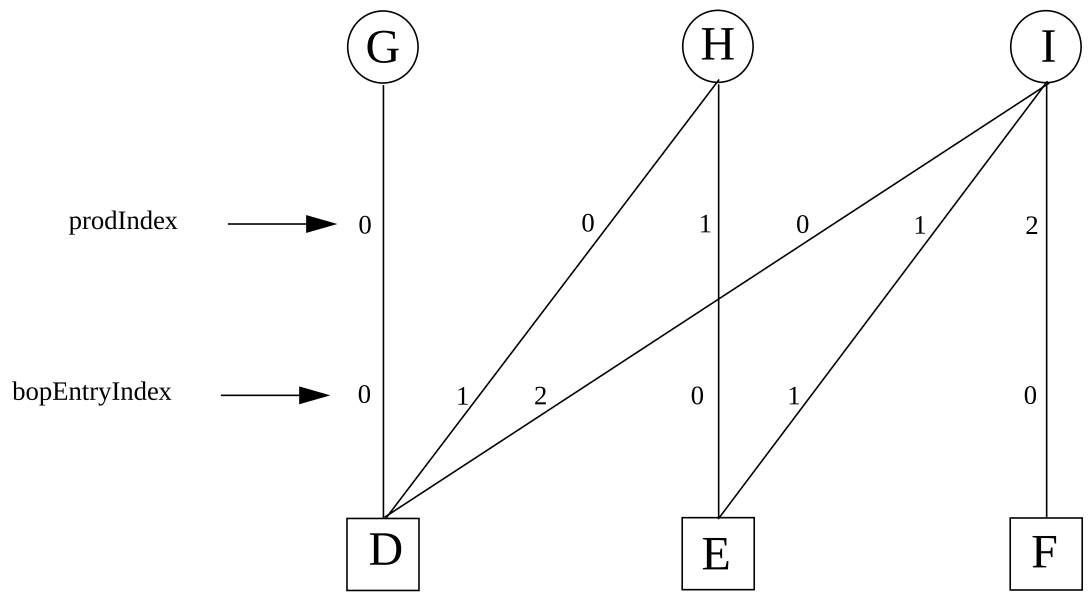
On return contains the bopEntryIndex of the specified producing BOP entry.

Error Conditions

- `partName` does not match the partName of an existing part.
- A `prodIndex` outside the range:

  $0 \leq$ `prodIndex` < number of producing BOP entries for the specified part. See “witGetPartNProducingBopEntries” on page 211.

FIGURE 21 Example use of witGetPartNProducingBopEntries and witGetPartProducingBopEntry



Example use of witGetPartNProducingBopEntries and witGetPartProducingBopEntry.

```
int    nProducingBopEntries;
int    prodIndex;
char * producingOperationName;
int    bopEntryIndex;

witGetPartNProducingBopEntries (
   theWitRun, "I", & nProducingBopEntries);

printf (
  "Part I is produced through the following %d "
  "BOP entries:\n\n",
  nProducingBopEntries);

for (prodIndex = 0;
     prodIndex < nProducingBopEntries;
     ++ prodIndex)
   {
   witGetPartProducingBopEntry (
      theWitRun,
      "I",
     prodIndex,
     & producingOperationName,
     & bopEntryIndex);

  printf (
    "   Operation %s, BOP entry #%d\n",
    producingOperationName,
    bopEntryIndex);

  free (producingOperationName);
  }
```

Given the problem shown above, this code generates the following output:

```
Part I is produced through the following 3 BOP entries:

    Operation D, BOP entry #2
    Operation E, BOP entry #1
    Operation F, BOP entry #0
```

witSetPartAttribute

```
witReturnCode witSetPartAttribute
(    WitRun * const theWitRun,
     const char * const partName,
     Type value );
```

`witSetPartAttribute` represents a group of functions for setting attribute values of a part. For more information on part attributes see “Part Attributes” on page 118.

The `witSetPartAttribute` functions are:

```
witReturnCode witSetPartAppData
(    WitRun * const theWitRun,
     const char * const partName,
     void * const appData );
witReturnCode witSetPartAsapPipOrder
(    WitRun * const theWitRun,
     const char * const partName,
     const witBoolean asapPipOrder );
witReturnCode witSetPartBoundedLeadTimes
(    WitRun * const theWitRun,
     const char * const partName,
     const witBoolean boundedLeadTimes );
witReturnCode witSetPartBuildAheadUB
(    WitRun * const theWitRun,
     const char * const partName,
     const int * buildAheadUB );
witReturnCode witSetPartBuildAsap
(    WitRun * const theWitRun,
     const char * const partName,
     const witBoolean buildAsap );
witReturnCode witSetPartBuildNstn
(    WitRun * const theWitRun,
     const char * const partName,
     const witBoolean buildNstn );
witReturnCode witSetPartPartName
(    WitRun * const theWitRun,
     const char * const partName,
     const char * const newName);
witReturnCode witSetPartPropRtg
(    WitRun * const theWitRun,
     const char * const partName,
     const witBoolean * propRtg );
witReturnCode witSetPartScrapAllowed
```

```
(    WitRun * const theWitRun,
     const char * const partName,
     const witBoolean scrapAllowed );
witReturnCode witSetPartScrapCost
(    WitRun * const theWitRun,
     const char * const partName,
     const float * const scrapCost );
witReturnCode witSetPartObjectStage
(    WitRun * const theWitRun,
     const char * const partName,
     const int objectStage );
witReturnCode witSetPartStockBounds
(    WitRun * const theWitRun,
     const char * const partName,
     const float * const hardLower,
     const float * const softLower,
     const float * const hardUpper );
witReturnCode witSetPartStockCost
(    WitRun * const theWitRun,
     const char * const partName,
     const float * const stockCost );
witReturnCode witSetPartSelForDel
(    WitRun * const theWitRun,
     const char * const partName,
     const witBoolean selForDel );
witReturnCode witSetPartSingleSource
(    WitRun * const theWitRun,
     const char * const partName,
     const witBoolean singleSource );
witReturnCode witSetPartSupplyVol
(    WitRun * const theWitRun,
     const char * const partName,
     const float * const supplyVol );
```

Description

`theWitRun`

Identifies the WIT problem to be used by this function.

`partName`

The partName of an existing part whose attribute value is to be modified.

`value`

The new value of the part attribute.

Usage notes

1. Setting some attributes will cause the state to change.   See “The State of a WitRun” on page 161.
2. `witSetPartPartName` changes the partName of the part to the value of the `newName` argument. Following a call to this function, any call to an API function that is to refer to this part must use the new partName. In addition to functions that explicitly refer to the part, this also includes any function that refers to a demand whose demanded part is the part whose partName was changed.

Error conditions

- A part with the specified `partName` has not been previously defined.
- Any violations of the requirements listed in “Part Attributes” on page 118.
- A bound set vector value which does not satisfy the requirements for a bound set. See “The State of a WitRun” on page 161.
- For stochastic implosion attributes, see Table 2 on page 27.
- In `witSetPartPartName`, the `newName` argument must be distinct from the partNames of all existing parts.

Exceptions to General Error Conditions

- In `witSetPartAppData`, the `appData` argument is allowed to be a NULL pointer.
- In `witSetPartStockBounds` , the `hardLower`, `softLower`, and `hardUpper` arguments are each allowed to be NULL pointers. In this case, a NULL pointer indicates that the corresponding bound set vector is to be unaltered.

Example #1

```
float fltv1[]    = {  1.,   3.,   5.  };
float infinity[] = { -1.0, -1.0, -1.0 };

witSetPartSupplyVol   (theWitRun, "prod1", fltv1);
witSetPartStockBounds (theWitRun, "prod1",
  NULL,      /* hard lower bounds are unchanged   */
  fltv1,     /* soft lower bounds = 1.,3.,5.      */
  infinity); /* hard upper bounds are infinite    */
```

This example assumes that there are 3 periods.

Example #2

```
witBoolean buildAsap;

witSetPartBuildAsap (theWR, "prod1",     WitTRUE);
witSetPartPartName  (theWR, "prod1",     "prod1-A");
```

```
witGetPartBuildAsap (theWR, "prod1-A", & buildAsap);

assert (buildAsap);
```

Demand Functions

The following functions allow the application program to access and modify data associated with a specific demand.

witAddDemand

```
witReturnCode witAddDemand
(    WitRun * const theWitRun,
     const char * const demandedPartName,
     const char * const demandName );
```

Description

This function creates a new demand with default attribute values.

`theWitRun`

Identifies the WIT problem to be used by this function.

`demandedPartName`

The demandedPartName of the new demand.

`demandName`

The demandName of the new demand.

Usage notes

1. The default values for demand attribute data are defined in “Demand Attributes” on page 126.

Error conditions

- Any violations of the requirements listed in “Demand Attributes” on page 126.
- The object iteration process must not be active.

Example

```
witReturnCode rc;
rc = witAddDemand(theWitRun,"prod1","demand1");
rc = witAddDemand(theWitRun,"raw1","demand2");
```

A demand is created for parts with names prod1 and raw1.

witGetDemandAttribute

```
witReturnCode witGetDemandAttribute
(    WitRun * const theWitRun,
     const char * const demandedPartName,
     const char * const demandName,
     Type value );
```

`witGetDemandAttribute` represents a group of functions for obtaining the value of demand attributes. For more information on demand attributes see “Demand Attributes” on page 126.

The `witGetDemandAttribute` functions are:

```
witReturnCode witGetDemandAppData
(    WitRun * const theWitRun,
     const char * const demandedPartName,
     const char * const demandName,
     void ** appData );
witReturnCode witGetDemandCumShipBounds
(     WitRun  * const  * theWitRun,
     const char * const demandedPartName,
     const char  * const demandName,
     float  * * hardLower )
     float  * * softLower )
     float  * * hardUpper );
 witReturnCode witGetDemandCumShipReward
(    WitRun * const theWitRun,
     const char  * const demandedPartName,
     const char  * const demandName,
     float * * cumShipReward );
witReturnCode witGetDemandDemandVol
(     WitRun  * const  * theWitRun,
     const char * const demandedPartName,
     const char  * const demandName,
     float  * * demandVol );
 witReturnCode witGetDemandFocusHorizon
(    WitRun * const theWitRun,
     const char  * const demandedPartName,
     const char  * const demandName,
     int * focusHorizon );
witReturnCode witGetDemandFssShipVol
(     WitRun  * const  * theWitRun,
     const char * const demandedPartName,
     const char  * const demandName,
     float  * * fssShipVol );
```

```
witReturnCode witGetDemandIntShipVols
(    WitRun * const theWitRun,
     const char  * const demandedPartName,
     const char  * const demandName,
     witBoolean * intShipVols );
witReturnCode witGetDemandLeadTimeUB
(    WitRun * const theWitRun,
     const char  * const demandedPartName,
     const char  * const demandName,
     int * * leadTimeUB );
witReturnCode witGetDemandPriority
(    WitRun * const theWitRun,
     const char  * const demandedPartName,
     const char  * const demandName,
     int * * priority );
witReturnCode witGetDemandSearchInc
(    WitRun * const theWitRun,
     const char  * const demandedPartName,
     const char  * const demandName,
     float * searchInc );
witReturnCode witGetDemandSelForDel
(    WitRun * const theWitRun,
     const char  * const demandedPartName,
     const char  * const demandName,
     witBoolean * selForDel );
witReturnCode witGetDemandShipLateAllowed
(    WitRun * const theWitRun,
     const char  * const demandedPartName,
     const char  * const demandName,
     witBoolean * shipLateAllowed );
witReturnCode witGetDemandShipLateUB
(    WitRun * const theWitRun,
     const char  * const demandedPartName,
     const char  * const demandName,
     int ** shipLateUB );
witReturnCode witGetDemandShipReward
(    WitRun * const theWitRun,
     const char  * const demandedPartName,
     const char  * const demandName,
     float ** shipReward );
```

```
witReturnCode witGetDemandShipVol
(    WitRun * const  * theWitRun,
     const char  * const demandedPartName,
     const char  * const demandName,
     float  * * shipVol );
```

Description

`theWitRun`

Identifies the WIT problem to be used by this function.

`demandedPartName`

The demandedPartName for the demand whose attribute value is to be returned (i.e., the partName of the demanded part).

`demandName`

The demandName for the demand whose attribute value is to be returned.

`value`

The location where the attribute value is returned.

Usage notes

1. When the type of the 4th parameter value is `float ** or int **`, then a vector with length `nPeriods` is returned. It is the application's responsibility to free the returned vector.

Error Conditions

- A demand with the specified demandedPartName and demandName must have been previously defined. In particular, this means that:
  - A part with the specified demandedPartName has been defined.
  - A demand with the specified demandedPartName has been defined for the specified demanded part.

Example

```
float * shipVol;
int nPeriods, t, focusHorizon;
witGetDemandShipVol( theWitRun, "prod1", "demand1",
             &shipVol );
witGetNPeriods( theWitRun, &nPeriods );
for ( t=0; t<nPeriods; t++ )
     printf( "shipVol[%d]=%f\n", t, shipVol[t] );
free( shipVol );
witGetDemandFocusHorizon( theWitRun, "prod1", "demand",
             &focusHorizon);
printf( "focusHorizon=%d\n", focusHorizon );
```

The per period individual shipment volume of demand "demand1" for part "prod1" is obtained and printed.The focus horizon for the same demand is obtained and printed.

witGetDemandExecVolPegging

```
witReturnCode witGetDemandExecVolPegging
(    WitRun * const     theWitRun,
     const char * const demandedPartName,
     const char * const demandName,
     const int          shipPeriod,
     int *              lenLists,
     char * * *         operationNameList,
     int * *            execPeriodList,
     float * *          peggedExecVolList);
```

Description

Retrieves the concurrent execVol pegging associated with a specific demand in a specific shipment period. See “Concurrent Pegging” on page 76. The application program specifies a demand and shipment period with the arguments `demandedPartName`, `demandName`, and `shipPeriod`. On return, for `0 ≤ i < (*lenLists)`, `(*operationNameList)[i]` identifies an operation, `(*execPeriodList)[i]` identifies an execution period, and `(*peggedExecVolList)[i]` is a pegged execVol. The pegged execVol is the total amount by which the execVol of the operation was increased in the execution period in order to increase the shipVol of the demand in the shipment period, since the last time the pegging was cleared.

`theWitRun`

Identifies the WIT problem to be used by this function.

`demandedPartName`

The demandedPartName for the demand whose execVol pegging is to be returned (i.e., the partName of the demanded part).

`demandName`

The demandName for the demand whose execVol pegging is to be returned.

`shipPeriod`

The shipment period whose execVol pegging is to be returned.

`lenLists`

On return, `(*lenLists)` is the number of pegging triples retrieved.

`operationNameList`

On return, for `0 ≤ i < (*lenLists),(*operationNameList)[i]` is the operationName of the operation for pegging triple #`i`.

`execPeriodList`

On return, for `0 ≤ i < (*lenLists), (*execPeriodList)[i]` is the execution period for pegging triple #`i`.

`peggedExecVolList`

On return, for `0 ≤ i < (*lenLists)`, `(*peggedExecVolList)[i]` is the pegged execVol for pegging triple #`i`.

Usage notes

- It is the responsibility of the application to free the returned vectors.

Error conditions

- A demand with the specified demandedPartName and demandName must have been previously defined. See also “Error Conditions” on page 222.
- `shipPeriod` must be in the range:

  0 ≤ `shipPeriod` < nPeriods
- The perfPegging global attribute must be TRUE.

Example

```
void prtExecVolPegging (
      WitRun *     theWitRun,
      const char * partName,
      const char * demandName,
      int          shipPer)
   {
   int      lenLists;
   char * * operationNameList;
   int *    execPeriodList;
   float *  peggedExecVolList;
   int      theIdx;

   witGetDemandExecVolPegging (
      theWitRun,
      partName,
      demandName,
      shipPer,
      & lenLists,
      & operationNameList,
      & execPeriodList,
      & peggedExecVolList);

   printf (
      "\nExecVol Pegging for Part %s, Demand %s, "
      "Period %d:\n\n",
      partName,
      demandName,
      shipPer);

   for (theIdx = 0; theIdx < lenLists; theIdx ++)
      printf (
         "   Operation %s, Period %d, ExecVol: %.0f\n",
         operationNameList[theIdx],
         execPeriodList   [theIdx],
         peggedExecVolList[theIdx]);

   for (theIdx = 0; theIdx < lenLists; theIdx ++)
      free (operationNameList[theIdx]);

   free (operationNameList);
   free (execPeriodList);
   free (peggedExecVolList);
   }
```

Then the function call:

```
prtExecVolPegging (theWitRun, "Prod12", "Cust7", 2);
```

might generate the following output:

```
ExecVol Pegging for Part Prod12, Demand Cust7, Period 2:

   Operation Build-Prod12, Period 2, ExecVol: 30
   Operation Build-Part17, Period 2, ExecVol: 19
   Operation Build-Part17, Period 1, ExecVol: 11
   Operation Build-Part43, Period 1, ExecVol: 60
```

witGetDemandExists

```
witReturnCode witGetDemandExists (
   WitRun * const     theWitRun,
   const char * const demandedPartName,
   const char * const demandName,
   witBoolean *       exists);
```

Description

This function allows the application to determine whether or not a specified demand exists in a WitRun.

`theWitRun`

Identifies the WIT problem to be used by this function.

`demandedPartName`

The partName of the demanded part of the demand being specified.

`demandName`

The demandName of the demand being specified.

`exists`

On return, (* exists) will be `WitTRUE` if a demand with the specified demandedPartName and specified demandName exists in the WitRun; otherwise (* exists) will be `WitFALSE`. If a part with the specified demandedPartName does not exist in the WitRun, (* exists) will be `WitFALSE`.

Example

```
witBoolean exists;

witGetDemandExists (
   theWitRun, "partA", "demandB", & exists);

if (exists)
     printf ("demandB for partA exists.\n");
else
     printf ("demandB for partA does not exist.\n");
```

vwitGetDemandSubVolPegging

```
witReturnCode witGetDemandSubVolPegging
(    WitRun * const     theWitRun,
     const char * const demandedPartName,
     const char * const demandName,
     const int          shipPeriod,
     int *              lenLists,
     char * * *         operationNameList,
     int * *            bomEntryIndexList,
     int * *            subsBomEntryIndexList,
     int * *            execPeriodList,
     float * *          peggedSubVolList);
```

Description

Retrieves the concurrent subVol pegging associated with a specific demand in a specific shipment period. See “Concurrent Pegging” on page 76. The application program specifies a demand and shipment period with the arguments `demandedPartName`, `demandName`, and `shipPeriod`. On return, for `0 ≤ i < (*lenLists)`, `(*operationNameList)[i]`, `(*bomEntryIndexList)[i]`, and `(*subsBomEntryIndexList)[i]` identify a substitute BOM entry, `(*execPeriodList)[i]` identifies an execution period, and `(*peggedSubVolList)[i]` is a pegged subVol. The pegged subVol is the total amount by which the subVol of the substitute was increased in the execution period in order to increase the shipVol of the demand in the shipment period, since the last time the pegging was cleared.

`theWitRun`

Identifies the WIT problem to be used by this function.

`demandedPartName`

The demandedPartName for the demand whose subVol pegging is to be returned (i.e., the partName of the demanded part).

`demandName`

The demandName for the demand whose subVol pegging is to be returned.

`shipPeriod`

The shipment period whose subVol pegging is to be returned.

`lenLists`

On return, `(*lenLists)` is the number of pegging triples retrieved.

`operationNameList`

On return, for `0 ≤ i < (*lenLists), (*operationNameList)[i]` is the consumingOperationName of the substitute BOM entry for pegging triple #`i`.

`bomEntryIndexList`

On return, for `0 ≤ i < (*lenLists)`, `(*bomEntryIndexList)[i]` is the bomEntryIndex of the substitute BOM entry for pegging triple #`i`.

`subsBomEntryIndexList`

On return, for `0 ≤ i < (*lenLists)`, `(*subsBomEntryIndexList)[i]` is the subsBomEntryIndex of the substitute BOM entry for pegging triple #`i`.

`execPeriodList`

On return, for `0 ≤ i < (*lenLists)`, `(*execPeriodList)[i]` is the execution period for pegging triple #`i`.

`peggedSubVolList`

On return, for `0 ≤ i < (*lenLists)`, `(*peggedSubVolList)[i]` is the pegged subVol for pegging triple #`i`.

Usage notes

- It is the responsibility of the application to free the returned vectors.

Error conditions

- A demand with the specified demandedPartName and demandName must have been previously defined. See also “Error Conditions” on page 222.
- `shipPeriod` must be in the range:

  0 ≤ `shipPeriod` < nPeriods
- The perfPegging global attribute must be TRUE.

Example

`witGetDemandSubVolPegging` works similarly to `witGetDemandExecVolPegging`. See “witGetDemandExecVolPegging” on page 224 for an example of its use.

witSetDemandAttribute

```
witReturnCode witSetDemandAttribute
(    WitRun * const theWitRun,
     const char * const demandedPartName,
     const char * const demandName,
     Type value );
```

`witSetDemandAttribute` represents a group of functions for setting attribute values of the demand identified by `partName` and `demandName`. For more information on demand attributes see “Demand Attributes” on page 126.

The `witSetDemandAttribute` functions are:

```
witReturnCode witSetDemandAppData
(    WitRun * const theWitRun,
     const char * const demandedPartName,
     const char * const demandName,
     void * const appData );
witReturnCode witSetDemandCumShipBounds
(    WitRun * const theWitRun,
     const char * const demandedPartName,
     const char * const demandName,
     const float * const hardLower,
     const float * const softLower,
     const float * const hardUpper );
witReturnCode witSetDemandCumShipReward
(    WitRun * const theWitRun,
     const char * const demandedPartName,
     const char * const demandName,
     const float * const cumShipReward );
witReturnCode witSetDemandDemandName
(    WitRun * const theWitRun,
     const char * const demandedPartName,
     const char * const demandName,
     const char * const newName );
witReturnCode witSetDemandDemandVol
(    WitRun * const theWitRun,
     const char * const demandedPartName,
     const char * const demandName,
     const float * const demandVol );
witReturnCode witSetDemandFocusHorizon
(    WitRun * const theWitRun,
     const char * const demandedPartName,
     const char * const demandName,
    const int focusHorizon );
witReturnCode witSetDemandFssShipVol
(   WitRun * const theWitRun,
    const char * const demandedPartName,
    const char * const demandName,
    const float * const fssShipVol );
witReturnCode witSetDemandIntShipVols
(   WitRun * const theWitRun,
    const char * const demandedPartName,
    const char * const demandName,
    const witBoolean intShipVols );
witReturnCode witSetDemandLeadTimeUB
(   WitRun * const theWitRun,
    const char * const demandedPartName,
    const char * const demandName,
    const int * const leadTimeUB );
witReturnCode witSetDemandPriority
(   WitRun * const theWitRun,
    const char * const demandedPartName,
    const char * const demandName,
    const int * const priority );
witReturnCode witSetDemandSearchInc
(   WitRun * const theWitRun,
    const char * const demandedPartName,
    const char * const demandName,
    const float searchInc );
witReturnCode witSetDemandSelForDel
(   WitRun * const theWitRun,
    const char * const demandedPartName,
    const char * const demandName,
    const witBoolean selForDel );
witReturnCode witSetDemandShipLateAllowed
(   WitRun * const theWitRun,
    const char * const demandedPartName,
    const char * const demandName,
    const witBoolean shipLateAllowed );
witReturnCode witSetDemandShipLateUB
(   WitRun * const theWitRun,
    const char * const demandedPartName,
    const char * const demandName,
    const int * shipLateUB );
witReturnCode witSetDemandShipReward
(   WitRun * const theWitRun,
    const char * const demandedPartName,
        const char * const demandName,
        const float * const shipReward );
witReturnCode witSetDemandShipVol
(       WitRun * const theWitRun,
        const char * const demandedPartName,
        const char * const demandName,
        const float * const shipVol );
```

Description

`theWitRun`

Identifies the WIT problem to be used by this function.

`demandedPartName`

The demandedPartName of the demand whose attribute value is to be modified (i.e., the partName of the demanded part).

`demandName`

The demandName of the demand whose attribute value is to be modified.

`value`

The new value of the demand attribute.

Usage notes

1. Setting some attributes will cause the state to change. See “The State of a WitRun” on page 161.
2. The attribute `fssShipVol` should normally be set after doing an implosion, and before obtaining the `FocusShortageVol`.
3. `witSetDemandDemandName` changes the demandName of the demand to the value of the `newName` argument. Following a call to this function, any call to an API function that is to refer to this demand must use the new demandName.

Error conditions

- A demand with the specified demandedPartName and demandName must have been previously defined. See also “Error Conditions” on page 222.
- Any violations of the requirements listed in “Demand Attributes” on page 126.
- A bound set vector value which does not satisfy the requirements for a bound set. See “The State of a WitRun” on page 161.
- In `witSetDemandDemandName`, the `newName` argument must be distinct from the demandNames of all existing demands associated with the same part.

Exceptions to General Error Conditions

- In `witSetDemandAppData`, the `appData` argument is allowed to be a NULL pointer.

- In `witSetDemandCumShipBounds` , the `hardLower`, `softLower`, and `hardUpper` arguments are each allowed to be NULL pointers. In this case, a NULL pointer indicates that the corresponding bound set vector is to be unaltered.

Example

```
witReturnCode rc;
float fltv1[]    = {  1.,   3.,   5.  };
int   intv1[]    = {  7,    8,    9   };

rc = witSetDemandPriority( theWitRun,"prod1", "demand1",
             intv1 );
rc = witSetDemandDemandVol ( theWitRun, "prod1",
             "demand1", fltv1 );
rc = witSetDemandCumShipBounds ( theWitRun,"prod1",
     "demand1",
     NULL, /* hard lower bounds are unchanged   */
     fltv1,/* soft lower bounds = 1.,3.,5.      */
     NULL);/* hard upper bounds are unchanged   */
```

This example assumes that there are 3 time periods.

Operation Functions

The following functions allow the application program to access and modify data associated with a specific operation.

witAddOperation

```
witReturnCode witAddOperation(
   WitRun * const theWitRun,
   const char * const operationName
);
```

Description

This function creates a new operation with default attribute values.

`theWitRun`

Identifies the WIT problem to be used by this function.

`operationName`

The operationName of the new operation.

Usage notes

1. The default values for operation attribute data are defined in “Operation Attributes” on page 131.

Error conditions

- Any violations of the requirements listed in “Operation Attributes” on page 131.
- The object iteration process must not be active.

Example

```
witReturnCode rc;
rc = witAddOperation(theWitRun,"test");
```

One operation named “test” is created and added to the WIT data structures.

witGetOperationAttribute

```
witReturnCode witGetOperationAttribute
(    WitRun * const theWitRun,
     const char * const operationName,
     Type value );
```

`witGetOperationAttribute` represents a group of functions for obtaining the value of operation attributes. For more information on operation attributes see “Operation Attributes” on page 131.

The `witGetOperationAttribute` functions are:

```
witReturnCode witGetOperationAppData
(    WitRun * const theWitRun,
     const char * const operationName,
     void ** appData );
witReturnCode witGetOperationExecBounds
(    WitRun * const theWitRun,
     const char * const operationName,
     float ** hardLower ,
     float ** softLower ,
     float ** hardUpper );
witReturnCode witGetOperationExecCost
(    WitRun * const theWitRun,
     const char * const operationName,
     float ** execCost );
witReturnCode witGetOperationExecPenalty
(    WitRun * const theWitRun,
     const char * const operationName,
     float * execPenalty );
witReturnCode witGetOperationExecVol
(    WitRun * const theWitRun,
     const char * const operationName,
     float ** execVol );
witReturnCode witGetOperationExecutable
(    WitRun * const theWitRun,
     const char * const operationName,
     witBoolean ** executable );
witReturnCode witGetOperationFssExecVol
(    WitRun * const theWitRun,
     const char * const operationName,
     float ** fssExecVol );
witReturnCode witGetOperationIncLotSize
(    WitRun * const theWitRun,
     const char * const operationName,
    float ** incLotSize );
witReturnCode witGetOperationIncLotSize2
(   WitRun * const theWitRun,
    const char * const operationName,
    float ** incLotSize2 );
witReturnCode witGetOperationIntExecVols
(   WitRun * const theWitRun,
    const char * const operationName,
    witBoolean * intExecVols );
witReturnCode witGetOperationLotSize2Thresh
(   WitRun * const theWitRun,
    const char * const operationName,
    float ** lotSize2Thresh );
witReturnCode witGetOperationMinLotSize
(   WitRun * const theWitRun,
    const char * const operationName,
    float ** minLotSize );
witReturnCode witGetOperationMinLotSize2
(   WitRun * const theWitRun,
    const char * const operationName,
    float ** minLotSize2 );
witReturnCode witGetOperationMrpExecVol
(   WitRun * const theWitRun,
    const char * const operationName,
    float ** mrpExecVol );
witReturnCode witGetOperationObjectStage
(   WitRun * const theWitRun,
    const char * const operationName,
    int * objectStage );
witReturnCode witGetOperationPipEnabled
(   WitRun * const theWitRun,
    const char * const operationName,
    witBoolean * pipEnabled );
witReturnCode witGetOperationPipRank
(   WitRun * const theWitRun,
    const char * const operationName,
    int * pipRank );
witReturnCode witGetOperationSelForDel
(   WitRun * const theWitRun,
    const char * const operationName,
    witBoolean * selForDel );
witReturnCode witGetOperationTwoLevelLotSizes
(   WitRun * const theWitRun,
    const char * const operationName,
     witBoolean * twoLevelLotSizes );
witReturnCode witGetOperationYieldRate
(    WitRun * const theWitRun,
     const char * const operationName,
     float ** yieldRate );
```

Description

`theWitRun`

Identifies the WIT problem to be used by this function.

`operationName`

The operationName of the operation whose attribute value is to be returned.

`value`

The location where the attribute value is returned.

Usage notes

1. When the type of the third parameter value is `float **`, `int **` or `witBoolean **`, then a vector with length `nPeriods` is returned. It is the responsibility of the application to free the returned vector.
2. Concerning `witGetOperationMrpExecVol` — meaningful values will be returned only if an WIT-MRP solution exists.
3. Concerning `witGetOperationExecutable` — the values are computed by WIT when it performs preprocessing of the data. If preprocessing has not already been performed when `witGetOperationExecutable` is called, WIT will perform preprocessing automatically. Since the preprocessing results are discarded when various input attributes are set, alternating calls to `witGetOperationExecutable` with calls to “`witSet...`” functions could result in repeated preprocessing and slow performance.
4. Concerning `witGetOperationFssExecVol` — this function causes WIT to compute the FSS if needed.

Error conditions

- An operation with the specified operationName has not been previously defined.
- Any violations of the requirements listed in “Operation Attributes” on page 131.
- Concerning `witGetOperationFssExecVol` — The WitRun must be in a postprocessed state. See “The State of a WitRun” on page 161.
- Also concerning `witGetOperationFssExecVol` — The current implosion solution must not be user-specified. See “User-Specified Solution” on page 37.
- For stochastic implosion attributes, see Table 2 on page 27.

Example

```
int nPeriods, t;
float * mls;
witGetOperationMinLotSize (theWitRun, "test", &mls);
witGetNPeriods (theWitRun, &nPeriods);
for (t = 0; t < nPeriods; t++)
    printf ("min lot size[%d] = %f\n", t, mls[t]);
free (mls);
```

MinLotSize of operation "test" is printed.

witGetOperationExists

```
witReturnCode witGetOperationExists
(    WitRun * const theWitRun,
     const char * const operationName,
     witBoolean * exists );
```

Description

This function allows the application to determine if a specified operation is defined in the WIT data structures.

`theWitRun`

Identifies the WIT problem to be used by this function.

`operationName`

The operationName of the operation to be tested to see if it has already been defined.

`exists`

Returns `WitTRUE` if an operation with the specified operationName has been defined, otherwise `WitFALSE`.

Example

```
witBoolean exists;
witGetOperationExists(theWitRun, "operationA", &exists );
if (exists)
     printf ("operationA has been defined.\n");
else
     printf ("operationA has not been defined.\n");
```

witGetOperationNBomEntries

```
witReturnCode witGetOperationNBomEntries
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     int * nBomEntries );
```

Description

This function returns the number of BOM entries in the BOM of the specified operation.

`theWitRun`

Identifies the WIT problem to be used by this function.

`consumingOperationName`

The name of the operation whose number of BOM entries is to be returned.

`nBomEntries`

On return this contains the number of BOM entries in BOM of the specified consuming operation.

Usage notes

1. Most BOM entry functions require a bomEntryIndex argument. This index can be obtained by calling `witGetOperationNBomEntries` for the consuming operation just after the BOM entry has been created. The bomEntryIndex of the newly created BOM entry is given by:
   `(* nBomEntries) - 1.`

   (It may be necessary to store this value for later use.)

   See also the example below.

Error Conditions

- `consumingOperationName` does not match the operationName of an existing operation.

Example

```
/*------------------------------------------------------*/
/* This function adds a BOM entry for the given         */
/* operationName and partName and sets its offset       */
/* to the given offset value.                           */
/*------------------------------------------------------*/

void addBomEntryWithOffset (
      WitRun *      theWitRun,
      const char *  operationName,
      const char *  partName,
      const float * offset)
   {
   int nBomEntries;
   int bomEntryIndex;

   witAddBomEntry (theWitRun, operationName, partName);

   witGetOperationNBomEntries (
        theWitRun,
        operationName,
      & nBomEntries);

   bomEntryIndex = nBomEntries - 1;

   witSetBomEntryOffset (
      theWitRun,
      operationName,
      bomEntryIndex,
      offset);
   }
```

witGetOperationNBopEntries

```
witReturnCode witGetOperationNBopEntries
(    WitRun * const theWitRun,
     const char * const producingOperationName,
     int * nBopEntries );
```

Description

This function returns the number of BOP entries in the BOP of the specified producing operation.

`theWitRun`

Identifies the WIT problem to be used by this function.

`producingOperationName`

The name of the operation whose number of BOP entries is to be returned.

`nBopEntries`

On return this contains the number of BOP entries in BOP of the specified producing operation.

Usage notes

1. Most BOP entry functions require a bopEntryIndex argument. This index can be obtained by calling `witGetOperationNBopEntries` for the producing operation just after the BOP entry has been created. The bopEntryIndex of the newly created BOP entry is given by:

   ```
   (* nBopEntries) - 1.
   ```

   (It may be necessary to store this value for later use.)

   See also the example below.

Error Conditions

- `producingOperationName` does not match the operationName of an existing operation.

Example

```
/*-----------------------------------------------------*/
/* This function adds a BOP entry for the given        */
/* operationName and partName and sets its offset      */
/* to the given offset value.                          */
/*-----------------------------------------------------*/

void addBopEntryWithOffset (
      WitRun *      theWitRun,
      const char *  operationName,
      const char *  partName,
      const float * offset)
   {
   int nBopEntries;
   int bopEntryIndex;

   witAddBopEntry (theWitRun, operationName, partName);

   witGetOperationNBopEntries (
        theWitRun,
        operationName,
      & nBopEntries);

   bopEntryIndex = nBopEntries - 1;

   witSetBopEntryOffset (
      theWitRun,
      operationName,
      bopEntryIndex,
      offset);
   }
```

witSetOperationAttribute

```
witReturnCode witSetOperationAttribute
(    WitRun * const theWitRun,
     const char * const operationName,
     Type value );
```

`witSetOperationAttribute` represents a group of functions for setting the value of operation attributes. For more information on operation attributes see “Operation Attributes” on page 131.

The `witSetOperationAttribute` functions are:

```
witReturnCode witSetOperationAppData
(    WitRun * const theWitRun,
     const char * const operationName,
     void * const appData );
witReturnCode witSetOperationExecBounds
(    WitRun * const theWitRun,
     const char * const operationName,
     const float * hardLower ,
     const float * softLower ,
     const float * hardUpper );
witReturnCode witSetOperationExecCost
(    WitRun * const theWitRun,
     const char * const operationName,
     const float * execCost );
witReturnCode witSetOperationExecPenalty
(    WitRun * const theWitRun,
     const char * const operationName,
     const float execPenalty );
witReturnCode witSetOperationExecVol
(    WitRun * const theWitRun,
     const char * const operationName,
     const float * execVol );
witReturnCode witSetOperationIncLotSize
(    WitRun * const theWitRun,
     const char * const operationName,
     const float * incLotSize );
witReturnCode witSetOperationIncLotSize2
(    WitRun * const theWitRun,
     const char * const operationName,
     const float * incLotSize2 );
witReturnCode witSetOperationIntExecVols
(    WitRun * const theWitRun,
     const char * const operationName,
     const witBoolean intExecVols );
witReturnCode witSetOperationLotSize2Thresh
(    WitRun * const theWitRun,
     const char * const operationName,
     const float * lotSize2Thresh );
witReturnCode witSetOperationMinLotSize
(    WitRun * const theWitRun,
     const char * const operationName,
     const float * minLotSize );
witReturnCode witSetOperationMinLotSize2
(    WitRun * const theWitRun,
     const char * const operationName,
     const float * minLotSize2 );
witReturnCode witSetOperationOperationName
(    WitRun * const theWitRun,
     const char * const operationName,
     const char * const newName );
witReturnCode witSetOperationObjectStage
(    WitRun * const theWitRun,
     const char * const operationName,
     const int objectStage );
witReturnCode witSetOperationPipEnabled
(    WitRun * const theWitRun,
     const char * const operationName,
     const witBoolean pipEnabled );
witReturnCode witSetOperationPipRank
(    WitRun * const theWitRun,
     const char * const operationName,
     const int pipRank );
witReturnCode witSetOperationSelForDel
(    WitRun * const theWitRun,
     const char * const operationName,
     const witBoolean selForDel );
witReturnCode witSetOperationTwoLevelLotSizes
(    WitRun * const theWitRun,
     const char * const operationName,
     const witBoolean twoLevelLotSizes );
witReturnCode witSetOperationYieldRate
(    WitRun * const theWitRun,
     const char * const operationName,
     const float * const yieldRate );
```

Description

`theWitRun`

Identifies the WIT problem to be used by this function.

`operationName`

The OperationName of an existing operation whose attribute value is to be modified.

`value`

The new value of the operation attribute.

Usage notes

1. Setting some attributes will cause the state to change. See "The State of a WitRun" on page 161.
2. `witSetOperationOperationName` changes the operationName of the operation to the value of the `newName` argument. Following a call to this function, any call to an API function that is to refer to this operation must use the new operationName. In addition to functions that explicitly refer to the operation, this also includes any function that refers to a BOM entry, substitute, or BOP entry corresponding to the operation whose operationName was changed.

Error conditions

- An operation with the specified operationName has not been previously defined.
- Any violations of the requirements listed in "Operation Attributes" on page 131.
- A bound set vector value which does not satisfy the requirements for a bound set. See "The State of a WitRun" on page 161.
- For stochastic implosion attributes, see Table 2 on page 27.
- In `witSetOperationOperationName`, the `newName` argument must be distinct from the operationNames of all existing operations.

Exceptions to General Error Conditions

- In `witSetOperationAppData`, the `appData` argument is allowed to be a NULL pointer.
- In `witSetOperationExecBounds` , the `hardLower`, `softLower`, and `hardUpper` arguments are each allowed to be NULL pointers. In this case, a NULL pointer indicates that the corresponding bound set vector is to be unaltered.

Example

```
witReturnCode rc;
float fltv1[]    = {  1.,   3.,   5.  };
int   fltv2[]    = {  .7,    .8,    .9   };
float infinity[] = { -1.0, -1.0, -1.0 };
```

```
rc = witSetOperationYieldRate ( theWitRun,"test", fltv2);
rc = witSetOperationExecBounds ( theWitRun,"test",
NULL,      / * hard lower bounds are unchanged   */
fltv1,     / * soft lower bounds = 1.,3.,5.      */
infinity); / * hard upper bounds are infinite    */
```

This example assumes that there are 3 time periods.

BOM Entry Functions

The following functions allow the application program to access and modify data associated with a specific BOM entry.

witAddBomEntry

```
witReturnCode witAddBomEntry
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const char * const consumedPartName );
```

Description

The function creates a new BOM entry with default attribute values.

`theWitRun`

Identifies the WIT problem to be used by this function.

`consumingOperationName`

The consumingOperationName of the new BOM entry.

`consumedPartName`

The consumedPartName of the new BOM entry.

Usage notes

1. The default values for BOM entry attribute data are defined in “BOM Entry Attributes” on page 138.

- The object iteration process must not be active.

Error conditions

- Any violations of the requirements listed in “BOM Entry Attributes” on page 138.

Example

```
witReturnCode rc;
rc = witAddBomEntry(theWitRun,"oper1","raw1");
rc = witAddBomEntry(theWitRun,"oper1","capacity1");
rc = witAddBomEntry(theWitRun,"oper1","raw1");
```

Three entries are added to the BOM for operation “oper1”.

witGetBomEntryAttribute

```
witReturnCode witGetBomEntryAttribute
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     Type value );
```

`witGetBomEntryAttribute` represents a group of functions for obtaining the value of BOM entry attributes. For more information on BOM entry attributes see “BOM Entry Attributes” on page 138.

The `witGetBomEntryAttribute` functions are:

```
 witReturnCode witGetBomEntryAppData
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     void ** appData );
witReturnCode witGetBomEntryConsRate
(    WitRun * const theWitRun,
     const char  * const consumingOperationName,
     const int bomEntryIndex,
     float ** consRate );
witReturnCode witGetBomEntryConsumedPart
(    WitRun * const theWitRun,
     const char  * const consumingOperationName,
     const int bomEntryIndex,
     char ** consumedPartName );
 witReturnCode witGetBomEntryEarliestPeriod
(    WitRun * const theWitRun,
     const char  * const consumingOperationName,
     const int bomEntryIndex,
     int * earliestPeriod );
witReturnCode witGetBomEntryExecPenalty
(    WitRun * const theWitRun,
     const char  * const consumingOperationName,
     const int bomEntryIndex,
     float * execPenalty );
witReturnCode witGetBomEntryImpactPeriod
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     int ** impactPeriod );
witReturnCode witGetBomEntryFalloutRate
(    WitRun * const theWitRun,
     const char  * const consumingOperationName,
     const int bomEntryIndex,
     float * falloutRate );
 witReturnCode witGetBomEntryLatestPeriod
(    WitRun * const theWitRun,
     const char  * const consumingOperationName,
     const int bomEntryIndex,
     int * latestPeriod );
 witReturnCode witGetBomEntryMandEC
(    WitRun * const theWitRun,
     const char  * const consumingOperationName,
     const int bomEntryIndex,
     witBoolean * mandEC );
witReturnCode witGetBomEntryOffset
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     float ** offset );
witReturnCode witGetBomEntryPropRtg
(    WitRun * const theWitRun,
     const char  * const consumingOperationName,
     const int bomEntryIndex,
     witBoolean ** propRtg );
witReturnCode witGetBomEntryRoutingShare
(    WitRun * const theWitRun,
     const char  * const consumingOperationName,
     const int bomEntryIndex,
     float ** routingShare );
witReturnCode witGetBomEntrySelForDel
(    WitRun * const theWitRun,
     const char  * const consumingOperationName,
     const int bomEntryIndex,
     witBoolean * selForDel );
witReturnCode witGetBomEntrySingleSource
(    WitRun * const theWitRun,
     const char  * const consumingOperationName,
     const int bomEntryIndex,
     witBoolean * singleSource );
```

Description

`theWitRun`

Identifies the WIT problem to be used by this function.

`consumingOperationName`

The consumingOperationName of the BOM entry whose attribute value is to be returned (i.e., the operationName of the consuming operation).

`bomEntryIndex`

The bomEntryIndex of the BOM entry whose attribute value is to be returned.

`value`

The location where the attribute value is returned.

Usage notes

1. For a suggestion on how to determine the bomEntryIndex of a BOM entry, see the usage note on page 241 and the example on page 242.
2. When the type of the 4th parameter value is `char **`, then a `char *` vector is returned. It is the application's responsibility to free the returned vector.
3. Concerning `witGetBomEntryImpactPeriod`—the values are computed by WIT when it performs preprocessing of the data. If preprocessing has not already been performed when `witGetBomEntryImpactPeriod` is called, WIT will perform preprocessing automatically. See note 3 on page 238 for the performance implications of this situation.

Error Conditions

- A BOM entry with the specified consumingOperationName and bomEntryIndex must have been previously defined. In particular, this means that:
  - An operation with the specified consumingOperationName has been defined.
  - `bomEntryIndex is` within the range:

    0 ≤ bom`EntryIndex` < NB

    where NB is the number of BOM entries for the consuming operation.

Example

```
char * consumedPartName;
float * consRate;
witGetBomEntryConsumedPart( theWitRun,
     "oper1", 1, &consumedPartName );
witGetBomEntryConsRate( theWitRun,
     "oper1", 1, &consRate );
printf( "oper1 consumes %s at rate %f in period 3.\n",
     consumedPartName, consRate[3] );
free( consumedPartName );
free( consRate );
```

witGetBomEntryNSubsBomEntries

```
witReturnCode witGetBomEntryNSubsBomEntries
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     int * nSubsBomEntries );
```

Description

This function returns the number of substitute BOM entries associated with a specified BOM entry.

`theWitRun`

Identifies the WIT problem to be used by this function.

`consumingOperationName`

The consumingOperationName of the BOM entry being specified.

`bomEntryIndex`

The bomEntryIndex of the BOM entry being specified.

`nSubsBomEntries`

On return this is the number of substitute BOM entries associated with the specified BOM entry, i.e., the number of substitute BOM entries that substitute for the specified BOM entry.

Usage notes

1. For a suggestion on how to determine the bomEntryIndex of a BOM entry, see the usage note on page 241 and the example on page 242.
2. Most substitute BOM entry functions require a subsBomEntryIndex argument. This index can be obtained by calling `witGetBomEntryNSubsBomEntries` for the replaced BOM entry just after the substitute has been created. The subsBomEntryIndex of the newly created substitute is given by:

   ```
   (* nSubsBomEntries) - 1.
   ```

   (It may be necessary to store this value for later use.)

   See also the example below.

Error Conditions

- A BOM entry with the specified consumingOperationName and bomEntryIndex must have been previously defined. See also “Error Conditions” on page 252.

Example

```
/*-----------------------------------------------------*/
/* This function adds a substitute for the given       */
/* operationName, bomEntryIndex, and partName and      */
/* sets its offset to the given offset value.          */
/*-----------------------------------------------------*/

void addSubsBomEntryWithOffset (
      WitRun *      theWitRun,
      const char *  operationName,
      int           bomEntryIndex,
      const char *  partName,
      const float * offset)
   {
   int nSubsBomEntries;
   int subsBomEntryIndex;

   witAddSubsBomEntry (
      theWitRun,
      operationName,
      bomEntryIndex,
      partName);

   witGetBomEntryNSubsBomEntries (
        theWitRun,
        operationName,
        bomEntryIndex,
      & nSubsBomEntries);

   subsBomEntryIndex = nSubsBomEntries - 1;

   witSetSubsBomEntryOffset (
      theWitRun,
      operationName,
      bomEntryIndex,
      subsBomEntryIndex,
      offset);
   }
```

witSetBomEntryAttribute

```
witReturnCode witSetBomEntryAttribute
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     Type value );
```

`witSetBomEntryAttribute` represents a group of functions for setting attribute values of the BOM entry identified by `consumingOperationName` and `bomEntryIndex`. For more information see “BOM Entry Attributes” on page 138.

The `witSetBomEntryAttribute` functions are:

```
witReturnCode witSetBomEntryAppData
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     void * const appData );
witReturnCode witSetBomEntryConsRate
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const float * consRate );
witReturnCode witSetBomEntryEarliestPeriod
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const int earliestPeriod );
witReturnCode witSetBomEntryExecPenalty
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const float execPenalty );
witReturnCode witSetBomEntryFalloutRate
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const float falloutRate );
witReturnCode witSetBomEntryLatestPeriod
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const int latestPeriod );
witReturnCode witSetBomEntryMandEC
```

```
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const witBoolean mandEC );
witReturnCode witSetBomEntryOffset
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const float * const offset);
witReturnCode witSetBomEntryPropRtg
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const witBoolean * propRtg );
witReturnCode witSetBomEntryRoutingShare
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const float * routingShare );
witReturnCode witSetBomEntrySelForDel
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const witBoolean selForDel );
witReturnCode witSetBomEntrySingleSource
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const witBoolean singleSource );
```

Description

`theWitRun`

Identifies the WIT problem to be used by this function.

`consumingOperationName`

The consumingOperationName of the BOM entry whose attribute value is to be modified (i.e., the operationName of the consuming operation).

`bomEntryIndex`

The bomEntryIndex of the BOM entry whose attribute value is to be modified.

`value`

The new value of the BOM entry attribute.

Usage notes

1. For a suggestion on how to determine the bomEntryIndex of a BOM entry, see the usage note on page 241 and the example on page 242.
2. Setting some attributes will cause the state to change.   See “The State of a WitRun” on page 161.

Error conditions

- A BOM entry with the specified consumingOperationName and bomEntryIndex must have been previously defined. See also “Error Conditions” on page 252.
- Any violations of the requirements listed in “BOM Entry Attributes” on page 138.

Exceptions to General Error Conditions

- In `witSetBomEntryAppData`, the `appData` argument is allowed to be a NULL pointer.

Example

```
witReturnCode rc;

rc =
  witSetBomEntryLatestPeriod(theWitRun,"oper1", 0, 1 );
rc =
  witSetBomEntryMandEC(theWitRun,"oper1", 0, WitTRUE );
rc =
  witSetBomEntryEarliestPeriod(theWitRun,"oper1", 2, 2 );
```

This example indicates that the last effective date for the first BOM entry (bomEntryIndex = 0) is period 1 and that this is due to a mandatory engineering change. In period 2, the third BOM entry becomes effective.

Substitute BOM Entry Functions

The following functions allow the application program to access and modify data associated with a specific substitute BOM entry.

witAddSubsBomEntry

```
witReturnCode witAddSubsBomEntry
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const char * const consumedPartName );
```

Description

The function creates a new substitute BOM entry with default attribute values.

`theWitRun`

Identifies the WIT problem to be used by this function.

`consumingOperationName`

The consumingOperationName of the new substitute BOM entry.

`bomEntryIndex`

The bomEntryIndex of the new substitute BOM entry.

`consumedPartName`

The consumedPartName of the new substitute BOM entry.

Usage notes

1. The default values for substitute BOM entry attribute data are defined in “Substitute BOM Entry Attributes” on page 143.

Error conditions

- Any violations of the requirements listed in “Substitute BOM Entry Attributes” on page 143.
- The object iteration process must not be active.

Example

```
witReturnCode rc;
rc = witAddSubsBomEntry(theWitRun,"oper1",2,"Raw2");
```

Raw2 can be substituted in the BOM for oper1 for the BOM entry with index 2.

witGetSubsBomEntryAttribute

```
witReturnCode witGetSubsBomEntryAttribute
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const int subsBomEntryIndex,
     Type value );
```

`witGetSubsBomEntryAttribute` represents a group of functions for obtaining the value of substitute BOM entry attributes. For more information on substitute BOM entry attributes see “Substitute BOM Entry Attributes” on page 143.

The `witGetSubsBomEntryAttribute` functions are:

```
witReturnCode witGetSubsBomEntryAppData
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const int subsBomEntryIndex,
     void ** appData );
witReturnCode witGetSubsBomEntryConsRate
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const int subsBomEntryIndex,
     float ** consRate );
witReturnCode witGetSubsBomEntryConsumedPart
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const int subsBomEntryIndex,
     char ** consumedPartName );
witReturnCode witGetSubsBomEntryEarliestPeriod
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const int subsBomEntryIndex,
     int * earliestPeriod );
witReturnCode witGetSubsBomEntryExecPenalty
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const int subsBomEntryIndex,
     float * execPenalty );
```

```
witReturnCode witGetSubsBomEntryExpAllowed
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const int subsBomEntryIndex,
     witBoolean * expAllowed );
witReturnCode witGetSubsBomEntryExpNetAversion
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const int subsBomEntryIndex,
     float * expNetAversion );
witReturnCode witGetSubsBomEntryFalloutRate
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const int subsBomEntryIndex,
     float * falloutRate );
witReturnCode witGetSubsBomEntryFssSubVol
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const int subsBomEntryIndex,
     float ** fssSubVol );
witReturnCode witGetSubsBomEntryImpactPeriod
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const int subsBomEntryIndex,
     int ** impactPeriod );
witReturnCode witGetSubsBomEntryIntSubVols
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const int subsBomEntryIndex,
     witBoolean * intSubVols );
witReturnCode witGetSubsBomEntryLatestPeriod
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const int subsBomEntryIndex,
     int * latestPeriod );
witReturnCode witGetSubsBomEntryMrpNetAllowed
(    WitRun * const theWitRun,
    const char * const consumingOperationName,
    const int bomEntryIndex,
    const int subsBomEntryIndex,
    witBoolean * mrpNetAllowed );
witReturnCode witGetSubsBomEntryMrpSubVol
(   WitRun * const theWitRun,
    const char * const consumingOperationName,
    const int bomEntryIndex,
    const int subsBomEntryIndex,
    float ** mrpSubVol );
witReturnCode witGetSubsBomEntryNetAllowed
(   WitRun * const theWitRun,
    const char * const consumingOperationName,
    const int bomEntryIndex,
    const int subsBomEntryIndex,
    witBoolean * netAllowed );
witReturnCode witGetSubsBomEntryOffset
(   WitRun * const theWitRun,
    const char * const consumingOperationName,
    const int bomEntryIndex,
    const int subsBomEntryIndex,
    float ** offset );
witReturnCode witGetSubsBomEntryRoutingShare
(   WitRun * const theWitRun,
    const char * const consumingOperationName,
    const int bomEntryIndex,
    const int subsBomEntryIndex,
    float ** routingShare );
witReturnCode witGetSubsBomEntrySelForDel
(   WitRun * const theWitRun,
    const char * const consumingOperationName,
    const int bomEntryIndex,
    const int subsBomEntryIndex,
    witBoolean * selForDel );
witReturnCode witGetSubsBomEntrySubCost
(   WitRun * const theWitRun,
    const char * const consumingOperationName,
    const int bomEntryIndex,
    const int subsBomEntryIndex,
    float * * subCost );
witReturnCode witGetSubsBomEntrySubVol
(   WitRun * const theWitRun,
    const char * const consumingOperationName,
    const int bomEntryIndex,
        const int subsBomEntryIndex,
        float * * subVol );
```

Description

`theWitRun`

Identifies the WIT problem to be used by this function.

`consumingOperationName`

The consumingOperationName of the substitute BOM entry whose attribute value is to be returned (i.e., the operationName of the consuming operation).

`bomEntryIndex`

The bomEntryIndex of the substitute BOM entry whose attribute value is to be returned.

`subsBomEntryIndex`

The subsBomEntryIndex of the substitute BOM entry whose attribute value is to be returned.

`value`

The location where the attribute value is returned.

Usage Notes

1. For a suggestion on how to determine the subsBomEntryIndex of a substitute, see the usage note on page 253 and the example on page 254.
2. Concerning `witGetSubsBomEntryConsumedPartName`, `witGetSubsBomEntrySubVol` — It is the responsibility of the application to free the returned vector.
3. Concerning `witGetSubsBomEntryFssSubVol` — this function causes WIT to compute the FSS if needed.
4. Concerning `witGetSubsBomEntryImpactPeriod`—the values are computed by WIT when it performs preprocessing of the data. If preprocessing has not already been performed when `witGetSubsBomEntryImpactPeriod` is called, WIT will perform preprocessing automatically. See note 3 on page 238 for the performance implications of this situation.

Error Conditions

- A substitute BOM entry with the specified consumingOperationName, bomEntryIndex, and subsBomEntryIndex must have been previously defined. In particular, this means that:
  - An operation with the specified consumingOperationName has been defined.
  - `bomEntryIndex is` within the range:

    0 ≤ bom`EntryIndex` < NB

    where NB is the number of BOM entries for the consuming operation.

  - `subsBomEntryIndex` is within the range:

    0 ≤ subsBom`EntryIndex` < NS

    where NS is the number of substitute BOM entries for the replaced BOM entry.
- Concerning `witGetSubsBomEntryFssSubVol` — The WitRun must be in a postprocessed state. See “The State of a WitRun” on page 161.
- Also concerning `witGetSubsBomEntryFssSubVol` — The current implosion solution must not be user-specified. See “User-Specified Solution” on page 37

Example

```
char * consumedPartName;
float * subVol;
int   nPeriods, t;
witGetSubsBomEntryConsumedPart ( theWitRun,
     "oper1",2,1,&consumedPartName );
printf("%s is consumed when executing oper1",
     consumedPartName );
free( consumedPartName );
witGetSubsBomEntrySubVol ( theWitRun,"oper1",2,1,
             &subVol);
witGetNPeriods(theWitRun,&nPeriods);
for ( t=0; t<nPeriods; t++ )
     printf("subVol[%d]=%f\n",t,subVol[t]);
free(prodVol);
```

The name of the part consumed by substitute number 1 of BOM entry 2 is obtained and printed. The execution volume for "oper1" due to the consumption of substitute number 1 of BOM entry 2 is obtained and printed on `stdout`.

witSetSubsBomEntryAttribute

```
witReturnCode witSetSubsBomEntryAttribute
( WitRun  * const theWitRun,
const char * const consumingOperationName,
const int bomEntryIndex,
const int subsBomEntryIndex,
Type value );
```

`witSetSubsBomEntryAttribute` represents a group of functions for setting attribute values of the substitute BOM entry identified by `consumingOperationName`, `bomEntryIndex` and `subsBomEntryIndex`. For more information on substitute BOM entry attributes see “Substitute BOM Entry Attributes” on page 143.

The `witSetBomEntryAttribute` functions are:

```
witReturnCode witSetSubsBomEntryAppData
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const int subsBomEntryIndex,
     void * const appData );
witReturnCode witSetSubsBomEntryConsRate
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const int subsBomEntryIndex,
     const float * consRate );
witReturnCode witSetSubsBomEntryEarliestPeriod
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const int subsBomEntryIndex,
     const int earliestPeriod );
witReturnCode witSetSubsBomEntryExecPenalty
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const int subsBomEntryIndex,
     const float execPenalty );
witReturnCode witSetSubsBomEntryExpAllowed
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const int subsBomEntryIndex,
    const witBoolean expAllowed );
witReturnCode witSetSubsBomEntryExpNetAversion
(   WitRun * const theWitRun,
    const char * const consumingOperationName,
    const int bomEntryIndex,
    const int subsBomEntryIndex,
    const float expNetAversion );
witReturnCode witSetSubsBomEntryFalloutRate
(   WitRun * const theWitRun,
    const char * const consumingOperationName,
    const int bomEntryIndex,
    const int subsBomEntryIndex,
    const float falloutRate );
witReturnCode witSetSubsBomEntryIntSubVols
(   WitRun * const theWitRun,
    const char * const consumingOperationName,
    const int bomEntryIndex,
    const int subsBomEntryIndex,
    const witBoolean intSubVols );
witReturnCode witSetSubsBomEntryLatestPeriod
(   WitRun * const theWitRun,
    const char * const consumingOperationName,
    const int bomEntryIndex,
    const int subsBomEntryIndex,
    const int latestPeriod );
witReturnCode witSetSubsBomEntryMrpNetAllowed
(   WitRun * const theWitRun,
    const char * const consumingOperationName,
    const int bomEntryIndex,
    const int subsBomEntryIndex,
    const witBoolean mrpNetAllowed );
witReturnCode witSetSubsBomEntryNetAllowed
(   WitRun * const theWitRun,
    const char * const consumingOperationName,
    const int bomEntryIndex,
    const int subsBomEntryIndex,
    const witBoolean netAllowed );
witReturnCode witSetSubsBomEntryOffset
(   WitRun * const theWitRun,
    const char * const consumingOperationName,
    const int bomEntryIndex,
    const int subsBomEntryIndex,
    const float * const offset );
witReturnCode witSetSubsBomEntryRoutingShare
```

```
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const int subsBomEntryIndex,
     const float * routingShare );
witReturnCode witSetSubsBomEntrySelForDel
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const int subsBomEntryIndex,
     const witBoolean selForDel );
witReturnCode witSetSubsBomEntrySubCost
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const int subsBomEntryIndex,
     const float * const subCost );
witReturnCode witSetSubsBomEntrySubVol
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const int subsBomEntryIndex,
     const float * const subVol );
```

Description

`theWitRun`

Identifies the WIT problem to be used by this function.

`consumingOperationName`

The consumingOperationName of the substitute BOM entry to be modified (i.e., the operationName of the consuming operation).

`bomEntryIndex`

The bomEntryIndex of the substitute BOM entry to be modified.

`subsBomEntryIndex`

The subsBomEntryIndex of the substitute BOM entry to be modified.

`value`

The new value of the substitute BOM entry attribute.

Usage notes

1. For a suggestion on how to determine the subsBomEntryIndex of a substitute, see the usage note on page 253 and the example on page 254.
2. Setting some attributes will cause the state to change. See “The State of a WitRun” on page 161.

Error conditions

- A substitute BOM entry with the specified consumingOperationName, bomEntryIndex, and subsBomEntryIndex must have been previously defined. See also “Error Conditions” on page 263.
- Any violations of the requirements listed in “Substitute BOM Entry Attributes” on page 143.
- `witSetSubsBomEntryOffset` must not be called if the global independentOffsets attribute is FALSE.

Exceptions to General Error Conditions

- In `witSetSubsBomEntryAppData`, the `appData` argument is allowed to be a NULL pointer.

Example

```
witReturnCode rc;
rc = witSetSubsBomEntryFalloutRate (theWitRun,
             "oper1", 2, 1, 0.1 );
```

The substitute BOM entry numbered 1 for the BOM entry numbered 2 of oper1 has the fallout rate set to 0.1.

BOP Entry Functions

The following functions allow the application program to access and modify data associated with a specific BOP entry.

witAddBopEntry

```
witReturnCode witAddBopEntry
(    WitRun * const theWitRun,
     const char * const producingOperationName,
     const char * const producedPartName );
```

Description

This function is used to indicate that a part is produced by an operation.

`theWitRun`

Identifies the WIT problem to be used by this function.

`producingOperationName`

The producingOperationName of the new BOP entry.

`producedPartName`

The producedPartName of the new BOP entry.

Usage notes

1. The default values for part attribute data are defined in “BOP Entry Attributes” on page 148.

Error conditions

- Any violations of the requirements listed in “BOP Entry Attributes” on page 148.
- The object iteration process must not be active.

Example

```
witReturnCode rc;
rc = witAddBopEntry(theWitRun,"op1","partA1");
rc = witAddBopEntry(theWitRun,"op1","partA2");
rc = witAddBopEntry(theWitRun,"op2","partB1");
rc = witAddBopEntry(theWitRun,"op2","partB2");
```

Four BOP entries are created and added to the WIT data structures.

witGetBopEntryAttribute

```
witReturnCode witGetBopEntryAttribute
(    WitRun * const theWitRun,
     const char * const  producingOperationName,
     const int bopEntryIndex,
     Type value );
```

`witGetBopEntryAttribute` represents a group of functions for obtaining the value of BopEntry attributes. For more information on BopEntry attributes see “BOP Entry Attributes” on page 148.

The `witGetBopEntryAttribute` functions are:

```
witReturnCode witGetBopEntryAppData
(    WitRun * const theWitRun,
     const char * const producingOperationName,
     const int bopEntryIndex,
     void ** appData );
witReturnCode witGetBopEntryEarliestPeriod
(    WitRun * const theWitRun,
     const char * const producingOperationName,
     const int bopEntryIndex,
     int * earliestPeriod );
witReturnCode witGetBopEntryExpAllowed
(    WitRun * const theWitRun,
     const char * const producingOperationName,
     const int bopEntryIndex,
     witBoolean * expAllowed );
witReturnCode witGetBopEntryExpAversion
(    WitRun * const theWitRun,
     const char * const producingOperationName,
     const int bopEntryIndex,
     float * expAversion );
witReturnCode witGetBopEntryImpactPeriod
(    WitRun * const theWitRun,
     const char * const producingOperationName,
     const int bopEntryIndex,
     int ** impactPeriod );
witReturnCode witGetBopEntryLatestPeriod
(    WitRun * const theWitRun,
     const char * const producingOperationName,
     const int bopEntryIndex,
     int * latestPeriod );
witReturnCode witGetBopEntryOffset
(    WitRun * const theWitRun,
     const char * const producingOperationName,
     const int bopEntryIndex,
     float ** offset );
witReturnCode witGetBopEntryProductRate
(    WitRun * const theWitRun,
     const char * const producingOperationName,
     const int bopEntryIndex,
     float ** productRate );
witReturnCode witGetBopEntryProducedPart
(    WitRun * const theWitRun,
     const char * const producingOperationName,
     const int bopEntryIndex,
     char ** producedPartName );
witReturnCode witGetBopEntryRoutingShare
(    WitRun * const theWitRun,
     const char * const producingOperationName,
     const int bopEntryIndex,
     float ** routingShare );
witReturnCode witGetBopEntrySelForDel
(    WitRun * const theWitRun,
     const char * const producingOperationName,
     const int bopEntryIndex,
     witBoolean * selForDel );
```

Description

`theWitRun`

Identifies the WIT problem to be used by this function.

`producingOperationName`

The producingOperationName of the BOP entry whose attribute value is to be returned (i.e., the operationName of the producing operation).

`bopEntryIndex`

The bopEntryIndex of the BOP entry whose attribute value is to be returned.

`value`

The location where the attribute value is returned.

Usage notes

1. For a suggestion on how to determine the bopEntryIndex of a BOP entry, see the usage note on page 243 and the example on page 244.
2. When the type of the 4th parameter value is char **, then a char * vector is returned. It is the applications responsibility to free the returned vector.
3. Concerning `witGetBopEntryImpactPeriod`—the values are computed by WIT when it performs preprocessing of the data. If preprocessing has not already been performed when `witGetBopEntryImpactPeriod` is called,

WIT will perform preprocessing automatically. See note 3 on page 238 for the performance implications of this situation.

Error conditions

- A BOP entry with the specified producingOperationName and bopEntryIndex must have been previously defined. In particular, this means that:
  - An operation with the specified producingOperationName has been defined.
  - `bopEntryIndex is` within the range:

    $0 \leq$ bop`EntryIndex` $<$ NB

    where NB is the number of BOP entries for the producing operation.

Example

```
char * producedPartName;
float * productRate;
witGetBopEntryProducedPartName( theWitRun,
     "operationA", 1, &producedPartName );
witGetBomEntryProductRate( theWitRun,
     "operationA", 1, &productRate );
printf( "OperationA produces %s at rate %f "
        " in period 2.\n",
     producedPartName, productRate[2] );
free( producedPartName );
free( productRate );
```

witSetBopEntryAttribute

```
witReturnCode witSetBopEntryAttribute
(    WitRun * const theWitRun,
     const char * const producingOperationName,
     const int bopEntryIndex,
     Type value );
```

`witSetBopEntryAttribute` represents a group of functions for setting the value of BopEntry attributes. For more information on BopEntry attributes see “BOP Entry Attributes” on page 148.

The `witSetBopEntryAttribute` functions are:

```
witReturnCode witSetBopEntryAppData
(    WitRun * const theWitRun,
     const char * const producingOperationName,
     const int bopEntryIndex,
     void * const appData );
witReturnCode witSetBopEntryEarliestPeriod
(    WitRun * const theWitRun,
     const char * const producingOperationName,
     const int bopEntryIndex,
     const int earliestPeriod );
witReturnCode witSetBopEntryExpAllowed
(    WitRun * const theWitRun,
     const char * const producingOperationName,
     const int bopEntryIndex,
     const witBoolean expAllowed );
witReturnCode witSetBopEntryExpAversion
(    WitRun * const theWitRun,
     const char * const producingOperationName,
     const int bopEntryIndex,
     const float expAversion );
witReturnCode witSetBopEntryLatestPeriod
(    WitRun * const theWitRun,
     const char * const producingOperationName,
     const int bopEntryIndex,
     const int latestPeriod );
witReturnCode witSetBopEntryOffset
(    WitRun * const theWitRun,
     const char * const producingOperationName,
     const int bopEntryIndex,
     const float * const offset );
witReturnCode witSetBopEntryProductRate
(    WitRun * const theWitRun,
     const char * const producingOperationName,
     const int bopEntryIndex,
     const float * productRate );
witReturnCode witSetBopEntryRoutingShare
(    WitRun * const theWitRun,
     const char * const producingOperationName,
     const int bopEntryIndex,
     const float * routingShare );
witReturnCode witSetBopEntrySelForDel
(    WitRun * const theWitRun,
     const char * const producingOperationName,
     const int bopEntryIndex,
     const witBoolean selForDel );
```

Description

`theWitRun`

Identifies the WIT problem to be used by this function.

`producingOperationName`

The producingOperationName of the BOP entry to be modified (i.e., the operationName of the producing operation).

`bopEntryIndex`

The bopEntryIndex of the BOP entry to be modified.

`value`

The new value of the BOM entry attribute.

Usage notes

1. For a suggestion on how to determine the bopEntryIndex of a BOP entry, see the usage note on page 243 and the example on page 244.
2. Setting some attributes will cause the state to change. See “The State of a WitRun” on page 161.

Error conditions

- A BOP entry with the specified producingOperationName and bopEntryIndex must have been previously defined. See also “Error conditions” on page 272.
- Any violations of the requirements listed in “BOP Entry Attributes” on page 148.

Exceptions to General Error Conditions

- In `witSetBopEntryAppData`, the `appData` argument is allowed to be a NULL pointer.

Example

```
witReturnCode rc;
rc = witSetBopEntryLatestPeriod(theWitRun,"op1", 0, 1);
rc = witSetBopEntryMandEC(theWitRun,"op1", 0, WitTRUE );
rc = witSetBopEntryEarliestPeriod(theWitRun,"op1",2,2);
```

This example indicates that the last effective date for the first BOP entry (bopEntryIndex = 0) is period 1 and that this is due to a mandatory engineering change.In period 2, the third BOP entry becomes effective.

Action Functions

witClearStochSoln

```
witReturnCode witClearStochSoln (
   WitRun * const theWitRun);
```

Description

Causes WIT to exit stochastic solution mode, while staying in stochastic mode. See “Stochastic Implosion” on page 21.

`theWitRun`

Identifies the WIT problem to be used by this function.

Error conditions

- stochMode must be TRUE.
- stochSolnMode must be TRUE.

witEqHeurAlloc

```
witReturnCode witEqHeurAlloc
(  WitRun * const             theWitRun,
   const int                  lenLists,
   const char * const * const demandedPartNameList,
   const char * const * const demandNameList,
   const int * const          shipPeriodList,
   const float * const        desIncVolList,
   float * *                  incVolList );
```

Description

Performs equitable heuristic allocation. See “Equitable Heuristic Allocation” on page 72. The arguments specify a list of “allocation targets”. Attempts to increase the shipVols of the specified demands in the specified shipment periods by as much as possible, up to the specified desired shipment volumes, subject to keeping the execution and shipment schedules feasible.

`theWitRun`

Identifies the WIT problem to be used by this function.

`lenLists`

The number of allocation targets on which equitable allocation is to be performed.

`demandedPartNameList`

For `0 ≤ i < lenLists`, `demandedPartNameList`[i] is the demandedPartName of the demand of the i-th allocation target.

`demandNameList`

For `0 ≤ i < lenLists`, `demandedNameList`[i] is the demandName of the demand of the i-th allocation target.

`shipPeriodList`

For `0 ≤ i < lenLists`, `shipPeriodList`[i] is the shipment period of the i-th allocation target.

`desIncVolList`

For `0 ≤ i < lenLists`, `desIncVolList`[i] is the desired incremental shipment volume of the i-th allocation target.

`incVolList`

On return, for `0 ≤ i < lenLists`, `(* incVolList)`[i] is the acheived incremental shipment volume of the i-th allocation target.

Usage Notes

1. It is the responsibility of the application to free the returned vector, `(* incVolList)`.

Error conditions

- lenLists must be ≥ 1.
- For `0 ≤ i < lenLists`, a demand identified by `demandedPartNameList`[i] and `demandNameList`[i] must have been previously defined. See also “Error Conditions” on page 222.
- For `0 ≤ i < lenLists`, `shipPeriodList[i]` must be in the range:

  0 ≤ `shipPeriod[i]` < nPeriods
- For `0 ≤ i < lenLists`, `desIncVolList[i]` must be ≥ 0.0.
- Duplicate allocation targets are not allowed, i.e, if `i ≠ j`, then it is not allowable to specify:

  `demandedPartNameList`[i] = `demandedPartNameList`[j]

  `demandNameList`[i] = `demandNameList`[j]

  and `shipPeriodList`[i] = `shipPeriodList`[j]

  simultaneuously.
- Heuristic allocation must be active.
- The twoWayMultiExec global attribute must be FALSE; when it is TRUE, `witEqHeurAllocTwme` should be used.

Example

```
witReturnCode rc;
int i;

const char * demandedPartNameList[] = {"Part1", "Part2"};

const char * demandList[] = {"Demand1", "Demand2"};

int shipPeriodList[] = {3, 1};

float desIncVolList[] = {100.0, 75.0};

float * incVolList;

witSetEquitability (theWitRun, 10);

rc = witStartHeurAlloc (theWitRun);
rc = witEqHeurAlloc (
   theWitRun,
   2,
   demandedPartNameList,
   demandNameList,
   shipPeriodList,
   desIncVolList,
   & incVolList)

for (i = 0; i < 2; ++ i)
   printf (
      "Demanded Part: %s\n"
      "Demand:        %s\n"
      "Desired  increment to shipVol[%d]: %.0f\n"
      "Acheived increment to shipVol[%d]: %.0f\n\n",
      demandedPartNameList[i],
      demandNameList[i],
      shipPeriodList[i],
      desIncVolList[i],
      shipPeriodList[i],
      incVolList[i]);

free (incVolList);

rc = witFinishHeurAlloc ( theWitRun);
```

This example attempts to ship 100 units of “part1” to “demand1” in period 3, and 75 units of “part2” to “demand2” in period 1, resolving resource conflicts at a ratio of 100 to 75 (approximately).

witEqHeurAllocTwme

```
witReturnCode witEqHeurAllocTwme
(  WitRun * const             theWitRun,
   const int                  lenLists,
   const char * const * const demandedPartNameList,
   const char * const * const demandNameList,
   const int * const          shipPeriodList,
   const float * const        desIncVolList,
   float * *                  incVolList,
   const witBoolean * const   asapMultiExecList);
```

Description

Performs equitable heuristic allocation in two-way multi-exec mode. See “Two-Way Multiple Execution Periods” on page 84. This function works in the same way as `witEqHeurAlloc`, except as indicated below. See “witEqHeurAlloc” on page 277.

`asapMultiExecList`

For `0 ≤ i < lenLists`, if `asapMultiExecList`[i] is TRUE, the initial multi-exec direction for the i-th allocation target will be ASAP ordering; if it is FALSE, the initial direction for the target will be NSTN ordering.

All other arguments

See “witEqHeurAlloc” on page 277.

Usage Notes

See “witEqHeurAlloc” on page 277.

Error conditions

- See “witEqHeurAlloc” on page 277.
- The twoWayMultiExec global attribute must be TRUE; when it is FALSE, `witEqHeurAlloc` should be used.

Example

```
witReturnCode rc;
int i;

const char * demandedPartNameList[] = {"Part1", "Part2"};

const char * demandList[] = {"Demand1", "Demand2"};
```

```
int shipPeriodList[] = {3, 1};

float desIncVolList[] = {100.0, 75.0};

float * incVolList;

witBoolean asapMultiExecList[] = {WitTRUE, WitFALSE};

witSetEquitability (theWitRun, 10);

rc = witStartHeurAlloc (theWitRun);

rc = witEqHeurAllocTwme (
   theWitRun,
   2,
   demandedPartNameList,
   demandNameList,
   shipPeriodList,
   desIncVolList,
   & incVolList,
   asapMultiExecList)

for (i = 0; i < 2; ++ i)
   printf (
      "Demanded Part: %s\n"
      "Demand:        %s\n"
      "Desired  increment to shipVol[%d]: %.0f\n"
      "Acheived increment to shipVol[%d]: %.0f\n\n",
      demandedPartNameList[i],
      demandNameList[i],
      shipPeriodList[i],
      desIncVolList[i],
      shipPeriodList[i],
      incVolList[i]);

free (incVolList);

rc = witFinishHeurAlloc ( theWitRun);
```

This example attempts to ship 100 units of “part1” to “demand1” in period 3, and 75 units of “part2” to “demand2” in period 1, resolving resource conflicts at a ratio of 100 to 75 (approximately). The initial multi-exec direction for "part1" will be ASAP ordering; the initial multi-exec direction for "part2" will be NSTN ordering.

witEvalObjectives

```
witReturnCode witEvalObjectives
( WitRun  * const theWitRun );
```

Description

Evaluates the objective function values for the currently defined solution. See “Evaluating the Objective Functions of a User-Specified Solution” on page 54.

`theWitRun`

Identifies the WIT problem to be used by this function.

Usage notes

1. Normally, this function should only be invoked when the current implosion solution is feasible. See “Testing the Feasibility of a User-Specified Solution” on page 38. If it is invoked when the implosion solution is infeasible, the objective function values may not be meaningful, and a warning is issued.
2. The first time this function is called for a given WitRun, it will take considerably more CPU time than subsequent calls to this function for the same WitRun. This is because the first call causes the LP model to be generated, while subsequent calls simply use the previously generated LP model. Also, changing any data attribute specified as "No" in Table 6 on page 162 will cause the LP model to be discarded, so that the next call to `witEvalObjectives` will cause the LP model to be re-generated (which consumes additional CPU time.)

Example

```
witReturnCode rc;
rc = witEvalObjectives ( theWitRun);
```

witFinishHeurAlloc

```
witReturnCode witFinishHeurAlloc
( WitRun  * const theWitRun );
```

Description

Concludes heuristic allocation, with post-processing. See “Heuristic Allocation” on page 70.

`theWitRun`

Identifies the WIT problem to be used by this function.

Error conditions

- Heuristic allocation must be active.

Example

See “Example” on page 287.

witHeurImplode

```
witReturnCode witHeurImplode
( WitRun  * const theWitRun );
```

Description

Performs the heuristic implosion. See “Heuristic Implosion” on page 13.

`theWitRun`

Identifies the WIT problem to be used by this function.

Usage notes

1. If `witPreprocess` has not been invoked, `witHeurImplode` will perform preprocessing.

Example

```
witReturnCode rc;
rc = witHeurImplode ( theWitRun);
```

witIncHeurAlloc

```
witReturnCode witIncHeurAlloc
(  WitRun * const     theWitRun,
   const char * const demandedPartName,
   const char * const demandName,
   const int          shipPeriod,
   const float        desIncVol,
   float *            incVol );
```

Description

Increments heuristic allocation. See “Heuristic Allocation without Equitable Allocation” on page 71. Attempts to increase the shipVol of the specified demand in shipPeriod by as much as possible, up to desIncVol, subject to keeping the execution and shipment schedules feasible.

`theWitRun`

Identifies the WIT problem to be used by this function.

`demandedPartName`

The demandedPartName for the demand for which heuristic allocation is to be incremented.

`demandName`

The demandName for the demand for which heuristic allocation is to be incremented.

`shipPeriod`

The period in which heuristic allocation is to be incremented.

`desIncVol`

The desired (i.e., maximum) amount by which the shipVol for the specified demand is to be increased in shipPeriod.

`incVol`

On return, contains the actual amount by which the shipVol for the specified demand was increased in shipPeriod.

Error conditions

- A demand with the specified demandedPartName and demandName must have been previously defined. See also “Error Conditions” on page 222.
- `shipPeriod` must be in the range:

  $0 \leq$ `shipPeriod` $<$ nPeriods
- `desIncVol` must be ≥ 0.0.
- Heuristic allocation must be active.
- The twoWayMultiExec global attribute must be FALSE; when it is TRUE, `witStartHeurAllocTwme` should be used.

Example

```
float incVol;
witReturnCode rc;

rc = witStartHeurAlloc ( theWitRun);

rc = witIncHeurAlloc (
   theWitRun,
   "prod1",
   "demand1",
   2,
   100.0,
   & incVol);

rc = witIncHeurAlloc (
   theWitRun,
   "prod1",
   "demand1",
   3,
   100.0 - incVol,
   & incVol);

rc = witFinishHeurAlloc ( theWitRun);
```

This example attempts to ship 100 units of “prod1” to “demand1” in period 2, and then attempts to ship any remaining units in period 3.

witIncHeurAllocTwme

```
witReturnCode witIncHeurAllocTwme
(  WitRun * const     theWitRun,
   const char * const demandedPartName,
   const char * const demandName,
   const int          shipPeriod,
   const float        desIncVol,
   float *            incVol,
   witBoolean         asapMultiExec );
```

Description

Increments heuristic allocation in two-way multi-exec mode. See “Two-Way Multiple Execution Periods” on page 84. This function works in the same way as `witIncHeurAlloc`, except as indicated below. See “witIncHeurAlloc” on page 286.

`asapMultiExec`

If TRUE, the initial multi-exec direction for the incremental allocation will be ASAP ordering; if FALSE, the initial direction will be NSTN ordering.

All other arguments

See “witIncHeurAlloc” on page 286.

Error conditions

- See “witIncHeurAlloc” on page 286
- The twoWayMultiExec global attribute must be TRUE; when it is FALSE, `witIncHeurAlloc` should be used.

Example

```
float incVol;
witReturnCode rc;

rc = witSetTwoWayMultiExec (theWitRun, WitTRUE);

rc = witStartHeurAlloc (theWitRun);

rc = witIncHeurAllocTwme (
   theWitRun,
   "prod1",
   "demand1",
   2,
   100.0,
   & incVol,
   WitTRUE);

rc = witFinishHeurAlloc (theWitRun);
```

This example attempts to ship 100 units of “prod1” to “demand1” in period 2, using ASAP ordering as the initial multi-exec direction.

witMrp

```
witReturnCode witMrp
( WitRun  * const theWitRun );
```

Description

Performs WIT-MRP. See “In two-way multi-exec mode (twoWayMultiExec == TRUE), special API functions must used to invoke heuristic allocation. Specifically, instead of witIncHeurAlloc, the function witIncHeurAllocTwme must be invoked and instead of witEqHeurAlloc, the function witEqHeurAllocTwme must be invoked. These two “Twme” functions have an additional argument which specifies the initial multi-exec direction. In witIncHeurAllocTwme, a single initial multi-exec direction is specified, while in witEqHeurAllocTwme, an initial multi-exec direction must be specified for each allocation target.” on page 85.

`theWitRun`

Identifies the WIT problem to be used by this function.

Usage notes

1. If `witPreprocess` has not been invoked, `witMrp` will perform preprocessing.

Example

```
witReturnCode rc;
rc = witMrp ( theWitRun);
```

witOptImplode

```
witReturnCode witOptImplode
( WitRun  * const theWitRun );
```

Description

Performs an optimizing implosion. See “Optimizing Implosion” on page 14.

`theWitRun`

Identifies the WIT problem to be used by this function.

Usage notes

1. This function is not available on all WIT implementations.
2. If `witPreprocess` has not been invoked, `witOptImplode` will perform preprocessing.
3. All LP Solver messages are written to the file defined by the solverLogFileName attribute. If a file with the same name already exists, it will be overwritten.

Error conditions

- Running out of memory. This error could occur for most WIT functions, however it is most likely to occur with `witOptImplode`. If this should happen, `witHeurImplode` could be used as an alternative since it requires significantly less memory.
- The problem must contain at least one part.

Example

```
witReturnCode rc;
rc = witOptImplode (theWitRun);
```

witShutDownHeurAlloc

```
witReturnCode witShutDownHeurAlloc
( WitRun  * const theWitRun );
```

Description

Terminates heuristic allocation, without post-processing. See “Heuristic Allocation” on page 70.

`theWitRun`

Identifies the WIT problem to be used by this function.

Error conditions

- Heuristic allocation must be active.

Example

See “Example” on page 287.

witStartHeurAlloc

```
witReturnCode witStartHeurAlloc
( WitRun  * const theWitRun );
```

Description

Initiates heuristic allocation. See “Heuristic Allocation” on page 70.

`theWitRun`

Identifies the WIT problem to be used by this function.

Usage notes

1. If `witPreprocess` has not been invoked, `witStartHeurAlloc` will perform preprocessing.

Example

See “Example” on page 287.

witStochImplode

```
witReturnCode witStochImplode (
   WitRun * const theWitRun);
```

Description

Performs stochastic implosion. See “Stochastic Implosion” on page 21.

`theWitRun`

Identifies the WIT problem to be used by this function.

Usage notes

1. All LP Solver messages are written to the file defined by the solverLogFileName attribute. If a file with the same name already exists, it will be overwritten.

Error conditions

- See the notes (“When `witStochImplode` is called,”) on page 28.

Example

See “Sample 5: newsVendor.C” on page 433.

File Input and Output Functions

witDisplayData

```
witReturnCode witDisplayData
(    WitRun * const theWitRun,
     const char * const fileName );
```

Description

Displays the input data associated with `theWitRun`. If preprocessing has been done, then additional information will be displayed. This function is useful in determining that the input data has been correctly passed to WIT.

`theWitRun`

Identifies the WIT problem to be used by this function.

`fileName`

The name of the file where the information is to be written. If either `WitSTDOUT` or `"stdout"` is specified, the file is written to `stdout`. If the `fileName` is `NULL` or the value of `mesgFileName`, then the current message file is used.

Usage Notes

1. If this function just creates an empty file, be sure informational message printing has not been turned off by using `witSetMesgTimesPrint`.
2. If the file needs to be opened, it will be opened with `fopen` using an access mode of `mesgFileAccessMode` and it will be closed before this function returns.
3. The format of the file name depends on the platform file system and implementation of `fopen`.

Error Conditions

- Failures reported by `fopen` and other file I/O operations.

Example

```
witReturnCode rc;
rc = witDisplayData (theWitRun, NULL );
rc = witDisplayData (theWitRun, WitSTDOUT );
rc = witDisplayData (theWitRun, "/tmp/displayData.wit" );
```

The display data is written three times, once each to the current WIT message file, `stdout`, and the file `/tmp/displayData.wit`.

witReadData

```
witReturnCode witReadData
(    WitRun * const theWitRun,
     const char * const fileName );
```

Description

This function reads a WIT Input Data file into `theWitRun`. This causes WIT to add new data objects (parts, demands, etc.) to `theWitRun` and/or to set the values of attributes belonging to the new and/or pre-existing objects, according to the instructions given in the file.

`theWitRun`

Identifies the WIT problem to be used by this function.

`fileName`

Name of file to be read. If either `WitSTDIN` or `"stdin"` is specified, then file is read from `stdin`.

Usage notes

1. See “Input Data File” on page 439 of Appendix B for the format of this file.
2. If the fileName is neither `WitSTDIN` nor `"stdin"`, the file is opened using the C function `fopen` with an access mode of “`r`”.
3. One way to use this function is to call it once for a WitRun, to completely define a WIT problem. Another way to use this function is to build up a WIT problem by calling `witReadData` multiple times with different files, defining different aspects of the problem, possibly interspersed with other API calls that define some aspects directly.

Error conditions

- The file must be in the correct format and specify valid data.
- If the file adds any data objects, the object iteration process must not be active.
- If `witReadData` either returns with a return code ≥ `WitSEVERE_RC` or throws a WitErrorExc, then no further calls to `witReadData` may be made, using any WitRun.
- Multi-threaded overlapping calls to `witReadData` are not allowed. See “Thread Safety” on page 160.

Example

```
witReturnCode rc;

rc = witReadData(theWitRun,"main.data");
rc = witSetWbounds(theWitRun,100000.0);
rc = witReadData(theWitRun,"supply.data");
```

A WIT problem is built up by reading the main data from main.data, setting wbounds to 100000.0, and then reading the supply data from supply.data. (supply.data might contain supplyVols for the parts defined in main.data.)

witWriteCriticalList

```
witReturnCode witWriteCriticalList
(    WitRun * const theWitRun,
     const char * const fileName,
     const witFileFormat fileFormat,
     const int maxListLength );
```

Description

Writes the list of critical parts and periods.

`theWitRun`

Identifies the WIT problem to be used by this function.

`fileName`

The name of the file where the information is to be written. If `either WitSTDOUT` or`"stdout"` is specified, the file is written to `stdout`. If the `fileName` is `NULL` or the value of `mesgFileName`, then the current message file is used.

`fileFormat`

Indicates the format of the output file. The choices are `WitBSV` (blank separated values) or `WitCSV` (comma separated values).

`maxListLength`

Upper bound on the number of critical parts which will be listed. If zero is specified then all critical parts will be listed.

Usage Notes

1. If this function just creates an empty file, be sure informational message printing has not been turned off by using `witSetMesgTimesPrint`.
2. If the file needs to be opened, it will be opened with `fopen` using an access mode of `mesgFileAccessMode` and it will be closed before this function returns.
3. The format of the file name depends on the platform file system and implementation of `fopen`.
4. When the fileFormat is `WitBSV`, then the format of the output record can be found in “Critical Parts List Output File” on page 465 of Appendix B.
5. When the fileFormat is `WitCSV` the same information is written except values are separated by commas and the message number is WIT0380I.
6. The comma separated values format (`WitCSV`) is intended to be imported by a spreadsheet. With this use, it may be desirable to turn off the printing of the message number with `witSetMesgPrintNumber`.
7. If the printing of the message number field has been turned off, then the fields in columns 0-8 will not be printed and all other fields will be left shifted by 9 columns.

Error Conditions

- Failures reported by `fopen` and other file I/O operations.

Example

```
witReturnCode rc;
rc = witWriteCriticalList ( theWitRun, NULL, WitBSV, 0 );
rc = witWriteCriticalList ( theWitRun, WitSTDOUT,
             WitBSV, 0 );
rc = witWriteCriticalList ( theWitRun,
             "/tmp/criticalList.wit", WitCSV, 10 );
```

The list of critical parts is written three times, once each to the current WIT message file, `stdout`, and the file `/tmp/criticalList.wit`. The first two times all critical parts are written with blanks separating values. The third time only the 10 most significant critical parts are written with commas separating the values.

witWriteData

```
witReturnCode witWriteData
(    WitRun * const theWitRun,
     const char * const fileName );
```

Description

Writes the input data currently stored in WIT data structures in a format which can be read by `witReadData`. The input data objects and attributes written are the same as those copied by `witCopyData.` See "witCopyData" on page 310 for more details.

`theWitRun`

Identifies the WIT problem to be used by this function.

`fileName`

The name of the file where the information is to be written. If either `WitSTDOUT` or `"stdout"` is specified, the file is written to `stdout`. If the `fileName` is `NULL` or the value of `mesgFileName`, then the current message file is used.

Usage Notes

1. If the file needs to be opened, it will be opened with `fopen` using an access mode of `mesgFileAccessMode` and it will be closed before this function returns.
2. The format of the file name depends on the platform file system and implementation of `fopen`.
3. By default, values of type float are written with a moderate numerical precision. To have `witWriteData` use greater numerical precision for float values, set the global boolean attribute highPrecisionWD to TRUE. (See "highPrecisionWD" on page 104.) Currently, `witWriteData` prints float values using a "`%.g`" format when highPrecisionWD is FALSE and a "`%.14g`" format when highPrecisionWD is TRUE. Since WIT stores all float values internally as doubles, this much precision could potentially be meaningful. In particular, setting highPrecisionWD to TRUE may be appropriate when using double precision API functions. (See "Double Precision Functions" on page 389)

Error Conditions

- Failures reported by `fopen` and other file I/O operations.

Example

```
witReturnCode rc;
rc = witWriteData ( theWitRun, NULL );
rc = witWriteData ( theWitRun, WitSTDOUT );
rc = witWriteData ( theWitRun, "/tmp/wit.data" );
```

The WIT data file is written three times, once each to the current WIT message file, `stdout`, and the file `/tmp/wit.data`.

witWriteExecSched

```
witReturnCode witWriteExecSched
(    WitRun * const theWitRun,
     const char * const fileName,
     const witFileFormat fileFormat );
```

Description

Writes the execution schedule, stating the level of execution of each operation in each period.

`theWitRun`

Identifies the WIT problem to be used by this function.

`fileName`

The name of the file where the information is to be written. If either `WitSTDOUT` or `"stdout"` is specified, the file is written to `stdout`. If the `fileName` is `NULL` or the value of `mesgFileName`, then the current message file is used.

`fileFormat`

Indicates the format of the output file. The choices are `WitBSV` (blank separated values) or `WitCSV` (comma separated values).

Usage Notes

1. If this function just creates an empty file, be sure informational message printing has not been turned off by using `witSetMesgTimesPrint`.
2. If the file needs to be opened, it will be opened with `fopen` using an access mode of `mesgFileAccessMode` and it will be closed before this function returns.
3. The format of the file name depends on the platform file system and implementation of `fopen`.
4. See “Execution Schedule Output File” on page 458 of Appendix B for the file formats.
5. If the printing of the message number field has been turned off, then the fields in columns 0-8 will not be printed and all other fields will be left shifted by 9 columns.
6. The messages written by `witWriteExecSched` do not wrap to the next line after writing 80 characters. These messages extend beyond 80 characters.
7. When the fileFormat is `WitCSV` the same information is written except values are separated by commas and the message number in the first section is WIT0377I and WIT0378I in the second section.
8. The comma separated values format (`WitCSV`) is intended to be imported by a spreadsheet. With this use, it may be desirable to turn off the printing of the message number with `witSetMesgPrintNumber`.

Error Conditions

- Failures reported by `fopen` and other file I/O operations.
- An unrecognized `fileFormat`.

Example

```
witReturnCode rc;
rc = witWriteExecSched ( theWitRun, NULL, WitBSV );
rc = witWriteExecSched ( theWitRun, WitSTDOUT, WitBSV );
rc = witSetMesgPrintNumber ( theWitRun, WitFALSE, 377,
               WitFALSE );
rc = witSetMesgPrintNumber ( theWitRun, WitFALSE, 378,
              WitFALSE );
rc = witWriteExecSched ( theWitRun,
              "/tmp/execSched.wit", WitCSV );
```

The execution schedule is written three times, once each to the current WIT message file, `stdout`, and the file `/tmp/execSched.wit`. The first two times values will be separated by blanks. The last time values will be separated by commas and message numbers for messages 377 and 378 will not be written.

witWriteReqSched

```
witReturnCode witWriteReqSched
(    WitRun * const theWitRun,
     const char * const fileName,
     const witFileFormat fileFormat );
```

Description

Writes the Requirements Schedule as computed by WIT-MRP.

`theWitRun`

Identifies the WIT problem to be used by this function.

`fileName`

The name of the file where the information is to be written. If either `WitSTDOUT` or `"stdout"` is specified, then the file is written to `stdout`. If the `fileName` is `NULL` or the value of `mesgFileName`, then the current message file is used.

`fileFormat`

Indicates the format of the output file. The choices are `WitBSV` (blank separated values) or `WitCSV` (comma separated values).

Usage Notes

1. If this function just creates an empty file, be sure informational message printing has not been turned off by using `witSetMesgTimesPrint`.
2. If the file needs to be opened, it will be opened with `fopen` using an access mode of `mesgFileAccessMode` and it will be closed before this function returns.
3. The format of the file name depends on the platform file system and implementation of `fopen`.
4. If the file format is `WitBSV`, then the format of the output file is in "Requirements Schedule Output File" on page 463 of Appendix B.
5. If the printing of the message number field has been turned off, then the fields in columns 0-8 will not be printed and all other fields will be left shifted by 9 columns.
6. The required volume written is the `reqVol` attribute of a part.
7. The messages written by `witWriteReqSched` do not wrap to the next line after writing 80 characters. These messages extend beyond 80 characters.
8. When the fileFormat is `WitCSV` the same information is written except values are separated by commas and the message number is WIT0379I.
9. The comma separated values format (`WitCSV`) is intended to be imported by a spreadsheet. With this use, it may be desirable to turn off the printing of the message number with `witSetMesgPrintNumber`.

Error Conditions

- Failures reported by `fopen` and other file I/O operations.
- An unrecognized `fileFormat`.

Example

```
witReturnCode rc;
rc = witWriteReqSched ( theWitRun, NULL, WitBSV );
rc = witWriteReqSched ( theWitRun, WitSTDOUT, WitBSV );
rc = witSetMesgPrintNumber ( theWitRun, WitFALSE, 379,
             WitFALSE );
rc = witWriteReqSched ( theWitRun,
             "/tmp/reqSched.wit", WitCSV);
```

The display data is written three times, once each to the current WIT message file, `stdout`, and the file `/tmp/reqSched.wit`. The first two times values will be separated by blanks. The last time values will be separated by commas and message numbers for message 379 will not be written.

witWriteShipSched

```
witReturnCode witWriteShipSched
(    WitRun * const theWitRun,
     const char * const fileName,
     const witFileFormat fileFormat );
```

Description

Writes the Shipment Schedule.

`theWitRun`

Identifies the WIT problem to be used by this function.

`fileName`

The name of the file where the information is to be written. If either `WitSTDOUT` or `"stdout"` is specified, the file is written to `stdout`. If the `fileName` is `NULL` or the value of `mesgFileName`, then the current message file is used.

`fileFormat`

Indicates the format of the output file. The choices are `WitBSV` (blank separated values) or `WitCSV` (comma separated values).

Usage Notes

1. If this function just creates an empty file, be sure informational message printing has not been turned off by using `witSetMesgTimesPrint`.
2. If the file needs to be opened, it will be opened with `fopen` using an access mode of `mesgFileAccessMode` and it will be closed before this function returns.
3. The format of the file name depends on the platform file system and implementation of `fopen`.
4. A shipment volume of zero is not written to the file.
5. If the `fileFormat` is `WitBSV`, the format of the output record is found in “Shipment Schedule Output File” on page 461 in Appendix B.
6. If the printing of the message number field has been turned off, then the fields in columns 0-8 will not be printed and all other fields will be left shifted by 9 columns.
7. The messages written by `witWriteShipSched` do not wrap to the next line after writing 80 characters. These messages extend beyond 80 characters.
8. When the fileFormat is `WitCSV` the same information is written except values are separated by commas and the message number is WIT0380I.
9. The comma separated values format (`WitCSV`) is intended to be imported by a spreadsheet. With this use, it may be desirable to turn off the printing of the message number with `witSetMesgPrintNumber`.

Error Conditions

- Failures reported by `fopen` and other file I/O operations.
- An unrecognized `fileFormat`.

Example

```
witReturnCode rc;
rc = witWriteShipSched ( theWitRun, NULL, WitBSV );
rc = witWriteShipSched ( theWitRun, WitSTDOUT, WitBSV );
rc = witSetMesgPrintNumber ( theWitRun, WitFALSE, 380,
             WitFALSE );
rc = witWriteShipSched ( theWitRun,
             "/tmp/shipSched.wit",WitCSV);
```

The shipment schedule is written three times, once each to the current WIT message file, `stdout`, and the file `/tmp/shipSched.wit.` The first two times values will be separated by blanks. The last time values will be separated by commas and message numbers for message 380 will not be written.

Utility Functions

witClearPegging

```
witReturnCode witClearPegging
( WitRun  * const theWitRun );
```

Description

This function clears the concurrent pegging, i.e., it deletes all currently existing pegging triples. (See “Concurrent Pegging” on page 76.)

`theWitRun`

The WitRun whose pegging is to be cleared.

Usage notes

- If `witGetDemandExecVolPegging` or `witGetDemandSubVolPegging` are called immediately after `witClearPegging`, they return empty lists (`*lenLists == 0`).

Error conditions

- The perfPegging global attribute must be TRUE.

Example

```
float incVol;

witIncHeurAlloc (
   theWitRun,
   "prod1",
   "cust1",
   3,
   100.0,
   & incVol);

witClearPegging (theWitRun);

witIncHeurAlloc (
   theWitRun,
   "prod1",
   "cust1",
   3,
   50.0,
   & incVol);

prtExecVolPegging (theWitRun, "prod1", "cust1", 3);
```

where `prtExecVolPegging` is as given on page 226. This code fragment prints the execVol pegging for the second call to `witIncHeurAlloc` only. The pegging for the first call is excluded.

witCopyData

```
witReturnCode witCopyData
( WitRun  * const dupWitRun,
  WitRun  * const origWitRun );
```

Description

This function copies the input data from `origWitRun` into `dupWitRun`. The data copied are:

- Global input attributes
- All parts and their input attributes
- All demands and their input attributes
- All operations and their input attributes
- All BOM entries and their input attributes
- All substitute BOM entries and their input attributes
- All BOP entries and their input attributes
- The PIP shipment sequence

The following input attributes are not copied:

- All message attributes.
- Operation execVol
- Substitute subVol
- Demand shipVol
- Demand fssShipVol
- optInitMethod
- solverLogFileName
- appData (all data objects)

(Copying the message attributes could have caused problems in some cases. The other attributes listed above are normally set by WIT.)

`dupWitRun`

The duplicate WitRun into which the input data is to be copied.

`origWitRun`

The original WitRun from which the input data is to be copied.

Usage notes

1. This function begins by invoking the internal equivalent of `witInitialize` on `dupWitRun`, which discards all previous data in `dupWitRun`, both input data and output data, except the message attributes, which are preserved. This puts `dupWitRun` into an unaccelerated, unpostprocessed state.

2. Since `witCopyData` calls an internal equivalent of `witInitialize`, a call to any function that must be preceeded by a call to `witInitialize` can legitimately preceeded by a call to `witCopyData` instead.
3. If `dupWitRun` and `origWitRun` are actually the same WitRun, this function does nothing and the input data, output data and state of `dupWitRun` are preserved.

Error conditions

- Must be preceeded by a call to `witInitialize (origWitRun)`.
- If witCopy`Data` either returns with a return code ≥ `WitSEVERE_RC` or throws a WitErrorExc, then neither `dupWitRun` nor `origWitRun` may be used in any further API function calls.

Exceptions to General Error Conditions

- Need not be preceeded by a call to `witInitialize (dupWitRun).`

Example

```
WitRun *  dupWitRun;
WitRun * origWitRun;

witNewRun (&  dupWitRun);
witNewRun (& origWitRun);

witSetMesgFileName  (dupWitRun, WitTRUE,  "dup.out");
witSetMesgFileName (origWitRun, WitTRUE, "orig.out);

witInitialize (origWitRun);
witReadData   (origWitRun, "wit.data");

witCopyData    (dupWitRun, origWitRun);
```

`dupWitRun` and `origWitRun` now contain the same input data. `dupWitRun` messages are written to file `dup.out` and `origWitRun` messages are written to `orig.out`.

witDeleteRun

```
witReturnCode witDeleteRun
( WitRun  * const theWitRun );
```

Description

This function frees the specified WitRun structure and the storage associated with that WIT problem.

`theWitRun`

Identifies the WitRun structure to be freed.

Usage notes

1. This function does not display any WIT messages.

Exceptions to General Error Conditions

- Need not be preceeded by a call to `witInitialize`.
- The `theWitRun` argument is allowed to be a NULL pointer. In this case, `witDeleteRun` does nothing.

Example

```
WitRun * theWitRun;
witDeleteRun( theWitRun );
```

The structure `theWitRun` is freed.

witFree

```
witReturnCode witFree
( void * mem );
```

Description

This function frees memory allocated by WIT. Typically an application can free memory that was allocated by WIT simply by calling `free`. However, in some situations, it may be necessary for WIT to free the memory, for example, when the application program is using a different runtime library than the one being used by WIT. In this case, `witFree` should be used.

`mem`

Identifies the memory to be freed.

Usage notes

1. This function does not display any WIT messages.

Exceptions to General Error Conditions

- Need not be preceeded by a call to `witInitialize`.
- The `mem` argument is allowed to be a NULL pointer. In this case, the function does nothing.

Example

```
float * sv;
witGetPartSupplyVol( wr, "P1", &sv );
witFree( sv );
```

The memory allocated by witGetPartSupplyVol and pointed to by sv is freed.

witGeneratePriorities

```
witReturnCode witGeneratePriorities
( WitRun  * const theWitRun );
```

Description

This function invokes WIT's "automatic priority" capability: WIT computes the demand priorities from the objective function attributes.

`theWitRun`

Identifies the WIT problem to be used by this function.

Usage notes

1. This function modifies the priority attribute of each demand in `theWitRun`.
2. This function does nothing on platforms where optimizing implosion is not supported.

Example

```
WitRun * theWitRun;
witGeneratePriorities( theWitRun );
```

The demand priority data is replaced with priorities computed from the objective function attribute data.

witGetExpCycle

```
witReturnCode witGetExpCycle (
   WitRun * const theWitRun,
   int *          lenLists,
   char * * *     partNameList,
   char * * *     operationNameList);
```

Description

This function searches for explodeable cycles in the complete BOM structure and retrieves one of them, if there are any. For information on explodeable cycles, note 2 on page 148.

`theWitRun`

Identifies the WIT problem to be used by this function.

`lenLists`

On return, `(*lenLists)` is the number of parts and the number of operations in the cycle, or 0, if there is no cycle.

`partNameList`

On return, for `0 ≤ i < (*lenLists)`, `(*partNameList)[i]` is the partName of part #`i` in the cycle.

`operationNameList`

On return, for `0 ≤ i < (*lenLists)`, `(*operationNameList)[i]` is the operationName of operation #`i` in the cycle.

Usage notes

1. When WIT finds an explodeable cycle during the execution of this function, it does not issue an error messagefor it.
2. If there are no explodeable cycles, then on return, `lenLists` = 0 and `partNameList` and `operationNameList` are empty.
3. If there is at least one explodeable cycle, then on return, the following conditions hold:
   - For `0 ≤ i < (*lenLists):`
     Part #i is produced by operation #i through an explodeable BOP entry.
   - For `0 ≤ i < (*lenLists)-1:`
     Operation #i consumes part #i+1 through a BOM entry or a substitute.
   - Operation #`(*lenLists)-1` consumes part #0 through a BOM entry or a substitute.
4. It is the responsibility of the application to free the returned vectors.

Example

The following function displays one explodeable cycle in `theWitRun`, if one exists; otherwise it indicates that no explodeable cycle exists.

```
void displayCycle (WitRun * theWitRun)
   {
   int      lenLists;
   char * * partNameList;
   char * * opnNameList;
   int      theIdx;

   witGetExpCycle (
        theWitRun,
      & lenLists,
      & partNameList,
      & opnNameList);

   if (lenLists == 0)
      {
      printf ("\n"
         "No explodeable cycles were found.\n");

      return;
      }

   printf ("\n"
      "The following explodeable cycle was found:\n");

   for (theIdx = 0; theIdx < lenLists; theIdx ++)
      printf (
         "   Part      %s\n"
         "   Operation %s\n",
         partNameList[theIdx],
         opnNameList [theIdx]);

   printf (
      "   Part      %s\n",
      partNameList[0]);

   for (theIdx = 0; theIdx < lenLists; theIdx ++)
      {
      free (partNameList[theIdx]);
      free (opnNameList [theIdx]);
      }

   free (partNameList);
   free (opnNameList);
   }
```

witInitialize

```
witReturnCode witInitialize
( WitRun  * const theWitRun );
```

Description

This function is used to establish the WIT environment for `theWitRun` specified. Every WIT application program must invoke this function at least once for each WitRun. This function can be used to reset `theWitRun` to its initial state.

`theWitRun`

Identifies the WIT problem to be used by this function.

Usage notes

1. This function will free most of the storage used by `theWitRun` and set `theWitRun` to its initial state, with the exception that attributes set by `witSetMesg`*`Attribute`* are not reset.
2. The first call to this function must be proceeded by a call to `witNewRun`, in order to provide the required non-NULL `theWitRun` argument. Between calling `witNewRun` and `witInitialize`, any of the `witSetMesg`*`Attribute`* functions may be called.

Exceptions to General Error Conditions

- Need not be preceeded by a call to `witInitialize`.

Example

```
witReturnCode rc;
WitRun * theWitRun;
rc = WitNewRun ( &theWitRun );
rc = witInitialize ( theWitRun );
```

The WIT environment for `theWitRun` is initialized.

witNewRun

```
witReturnCode witNewRun
( WitRun  * * theWitRun );
```

Description

This function allocates and returns a pointer to a WitRun structure. The returned `theWitRun` pointer identifies the WIT problem and is the first parameter to all WIT API functions.

`theWitRun`

Identifies where the pointer to the allocated WitRun structure is to be stored.

Usage notes

1. This function does not display any WIT messages.

Exceptions to General Error Conditions

- Need not be preceeded by a call to `witInitialize`.

Example

```
WitRun * witRun1;
WitRun * witRun2;
witNewRun( &witRun1 );
witNewRun( &witRun2 );
witInitialize( witRun1 );
witInitialize( witRun2 );
```

There are two active WIT problems identified by `witRun1` and `witRun2`.

witPostprocess

```
witReturnCode witPostprocess
( WitRun  * const theWitRun );
```

Description

This function changes the state of the WitRun to be postprocessed. See “The State of a WitRun” on page 161.

`theWitRun`

Identifies the WIT problem to be used by this function.

Usage notes

1. `witPostprocess` is automatically invoked by API functions that must put the WitRun into a postprocessed state. See “The State of a WitRun” on page 161. Thus it is not usually necessary for the application to explicitly invoke this function.

Example

```
witReturnCode rc;
rc = witPostprocess(theWitRun);
```

witPreprocess

```
witReturnCode witPreprocess
( WitRun  * const theWitRun );
```

Description

Before WIT does implosion or WIT-MRP, it performs certain preprocessing of the data. This function causes that preprocessing to be performed immediately. It is normally not necessary for the application to explicitly invoke this function.

`theWitRun`

Identifies the WIT problem to be used by this function.

Usage notes

1. When an API function requiring preprocessing is invoked, if preprocessing has not been done, then the API function automatically invokes `witPreprocess`.
2. This function causes `witDisplayData` to provide additional information.

Error conditions

- Preprocessing checks the ranges of the input data. Any errors found are reported.
- Preprocessing checks for illegal cycles in the complete BOM structure. For more details, see note 2 on page 148 .

Example

```
witReturnCode rc;
rc = witPreprocess(theWitRun);
```

witPurgeData

```
witReturnCode witPurgeData
( WitRun  * const theWitRun );
```

Description

This function causes WIT to do a purge. (See “Object Deletion” on page 33.) It deletes all data objects whose selForDel attribute is TRUE, as well as all data objects that have one or more prerequisites that are being deleted.

`theWitRun`

Identifies the WIT problem to be used by this function

Usage notes

1. Chapter 2 states that the bomEntryIndex attribute of a BOM entry is “the number of existing BOM entries for the consuming operation that were created before the current one”. Thus the bomEntryIndex of a BOM entry will be decreased by 1 whenever an earlier-created BOM entry for the same consuming operation is deleted. A similar situation occurs for the subsBomEntryIndex of a substitute BOM entry and for the bopEntryIndex of a BOP entry.

Error conditions

- The object iteration process must not be active.

Example

```
WitRun * theWitRun;

WitNewRun     (& theWitRun);
witInitialize   (theWitRun);

witAddPart      (theWitRun, "A", WitMATERIAL);
witAddOperation (theWitRun, "B");
witAddBomEntry  (theWitRun, "B", "A");
witAddDemand    (theWitRun, "A", "C");
witAddDemand    (theWitRun, "A", "D");

witSetOperationSelForDel (theWitRun, "B", WitTRUE);
witSetDemandSelForDel    (theWitRun, "A", "C", WitTRUE);

witPurgeData (theWitRun);
```

Operation B and demand C for part A will be deleted, because they were selected for deletion. The BOM entry from operation B to part A will also be deleted, because operation B is being deleted and it’s a prerequisite for this

BOM entry. Part A and demand D for part A will not be deleted, because they were not selected for deletion and have no prerequisites that are being deleted.

Input Data Copying Functions

The following functions copy the input data attributes from one data object into another data object of the same type.

witCopyBomEntryData

```
witReturnCode witCopyBomEntryData (
   WitRun * const     dupWitRun,
   const char * const dupOperationName,
   const int          dupBomEntryIndex,
   WitRun * const     origWitRun,
   const char * const origOperationName,
   const int          origBomEntryIndex);
```

Description

This function copies the input data attributes from one BOM entry into another BOM entry. The two BOM entries are referred to as “the original BOM entry” and “the duplicate BOM entry”. They may both belong to the same WitRun or to two different WitRuns.

`dupWitRun`

The WitRun to which the duplicate BOM entry belongs.

`dupOperationName`

The consumingOperationName of the duplicate BOM entry.

`dupBomEntryIndex`

The bomEntryIndex of the duplicate BOM entry.

`origWitRun`

The WitRun to which the original BOM entry belongs.

`origOperationName`

The consumingOperationName of the original BOM entry.

`origBomEntryIndex`

The bomEntryIndex of the original BOM entry.

Usage notes

1. Input data attributes (e.g. offset) are copied; attributes computed by WIT (e.g. impactPeriod) are not copied.
2. The following input attribute is not copied:
   - appData

Error conditions

- Must be preceeded by a call to `witInitialize (origWitRun)`.

- If `witCopyBomEntryData` either returns with a return code ≥ `WitSEVERE_RC` or throws a WitErrorExc, then neither `dupWitRun` nor `origWitRun` may be used in any further API function calls.
- The following global attributes must have the same value for `dupWitRun` and `origWitRun`:
  - nPeriods
  - independentOffsets
- `dupOperationName` and `dupBomEntryIndex` must identify an existing BOM entry in `dupWitRun`.
- `origOperationName` and `origBomEntryIndex` must identify an existing BOM entry in `origWitRun`.
- The duplicate and original BOM entries must not be the same BOM entry in the same WitRun. Thus either `dupWitRun` ≠ `origWitRun`, or `dupOperationName` ≠ `origOperationName`, or `dupBomEntryIndex ≠ origBomEntryIndex`.

Example

```
WitRun * origWitRun;
WitRun *  dupWitRun;

witNewRun    (& origWitRun);
witInitialize  (origWitRun);
witReadData    (origWitRun, "original.data");

witNewRun     (& dupWitRun);
witInitialize   (dupWitRun);
witAddPart      (dupWitRun, "A", WitMATERIAL);
witAddOperation (dupWitRun, "B");
witAddBomEntry  (dupWitRun, "B", "A");

witCopyBomEntryData (
    dupWitRun, "B", 0,
   origWitRun, "Z", 5);
```

After this code is executed, the BOM entry for operation B, index 0 in `dupWitRun` will have the same input data attribute values as operation Z, index 5 in `origWitRun`.

witCopyBopEntryData

```
witReturnCode witCopyBopEntryData (
   WitRun * const     dupWitRun,
   const char * const dupOperationName,
   const int          dupBopEntryIndex,
   WitRun * const     origWitRun,
   const char * const origOperationName,
   const int          origBopEntryIndex);
```

Description

This function copies the input data attributes from one BOP entry into another BOP entry. The two BOP entries are referred to as “the original BOP entry” and “the duplicate BOP entry”. They may both belong to the same WitRun or to two different WitRuns.

`dupWitRun`

The WitRun to which the duplicate BOP entry belongs.

`dupOperationName`

The producingOperationName of the duplicate BOP entry.

`dupBopEntryIndex`

The bopEntryIndex of the duplicate BOP entry.

`origWitRun`

The WitRun to which the original BOP entry belongs.

`origOperationName`

The producingOperationName of the original BOP entry.

`origBopEntryIndex`

The bopEntryIndex of the original BOP entry.

Usage notes

1. Input data attributes (e.g. offset) are copied; attributes computed by WIT (e.g. impactPeriod) are not copied.
2. The following input attribute is not copied:
   - appData

Error conditions

- Must be preceeded by a call to `witInitialize (origWitRun)`.
- If `witCopyBopEntryData` either returns with a return code ≥ `WitSEVERE_RC` or throws a WitErrorExc, then neither `dupWitRun` nor `origWitRun` may be used in any further API function calls.
- The following global attributes must have the same value for `dupWitRun` and `origWitRun`:
  - nPeriods

  - independentOffsets
- `dupOperationName` and `dupBopEntryIndex` must identify an existing BOP entry in `dupWitRun`.
- `origOperationName` and `origBopEntryIndex` must identify an existing BOP entry in `origWitRun`.
- The duplicate and original BOP entries must not be the same BOP entry in the same WitRun. Thus either `dupWitRun` ≠ `origWitRun`, or `dupOperationName` ≠ `origOperationName,or` `dupBopEntryIndex` ≠ `origBopEntryIndex`.

Example

This function is similar to `witCopyBomEntryData`. See the example on page 324.

witCopyDemandData

```
witReturnCode witCopyDemandData (
   WitRun * const     dupWitRun,
   const char * const dupPartName,
   const char * const dupDemandName,
   WitRun * const     origWitRun,
   const char * const origPartName,
   const char * const origDemandName);
```

Description

This function copies the input data attributes from one demand into another demand. The two demands are referred to as “the original demand” and “the duplicate demand”. They may both belong to the same WitRun or to two different WitRuns.

`dupWitRun`

The WitRun to which the duplicate demand belongs.

`dupPartName`

The demandedPartName of the duplicate demand.

`dupDemandName`

The demandName of the duplicate demand.

`origWitRun`

The WitRun to which the original demand belongs.

`origPartName`

The demandedPartName of the original demand.

`origDemandName`

The demandName of the original demand.

Usage notes

1. The following input attributes are not copied:
   - appData
   - shipVol
   - fssShipVol

Error conditions

- Must be preceeded by a call to `witInitialize (origWitRun)`.
- If `witCopyDemandData` either returns with a return code ≥ `WitSEVERE_RC` or throws a WitErrorExc, then neither `dupWitRun` nor `origWitRun` may be used in any further API function calls.
- The following global attributes must have the same value for `dupWitRun` and `origWitRun`:
  - nPeriods

  - independentOffsets
- `dupPartName` and `dupDemandName` must identify an existing demand in `dupWitRun`.
- `origPartName` and `origDemandName` must identify an existing demand in `origWitRun`.
- The duplicate and original demands must not be the same demand in the same WitRun. Thus either `dupWitRun` ≠ `origWitRun`, or `dupPartName` ≠ `origPartName`, or `dupDemandName` ≠ `origDemandName`.

Example

This function is similar to `witCopyBomEntryData`. See the example on page 324.

witCopyOperationData

```
witReturnCode witCopyOperationData (
   WitRun * const     dupWitRun,
   const char * const dupOperationName,
   WitRun * const     origWitRun,
   const char * const origOperationName);
```

Description

This function copies the input data attributes from one operation into another operation. The two operations are referred to as “the original operation” and “the duplicate operation”. They may both belong to the same WitRun or to two different WitRuns.

`dupWitRun`

The WitRun to which the duplicate operation belongs.

`dupOperationName`

The operationName of the duplicate operation.

`origWitRun`

The WitRun to which the original operation belongs.

`origOperationName`

The operationName of the original operation.

Usage notes

1. Input data attributes (e.g. execCost) are copied; attributes computed by WIT (e.g. executable) are not copied.
2. The following input attributes are not copied:
   - appData
   - execVol

Error conditions

- Must be preceeded by a call to `witInitialize (origWitRun)`.
- If `witCopyOperationData` either returns with a return code ≥ `WitSEVERE_RC` or throws a WitErrorExc, then neither `dupWitRun` nor `origWitRun` may be used in any further API function calls.
- The following global attributes must have the same value for `dupWitRun` and `origWitRun`:
  - nPeriods
  - independentOffsets
- `dupOperationName` must identify an existing operation in `dupWitRun`.
- `origOperationName` must identify an existing operation in `origWitRun`.

- The duplicate and original operations must not be the same operation in the same WitRun. Thus either `dupWitRun` ≠ `origWitRun`, or `dupOperationName` ≠ `origOperationName`.

Example

This function is similar to `witCopyBomEntryData`. See the example on page 324.

witCopyPartData

```
witReturnCode witCopyPartData (
   WitRun * const     dupWitRun,
   const char * const dupPartName,
   WitRun * const     origWitRun,
   const char * const origPartName);
```

Description

This function copies the input data attributes from one part into another part. The two parts are referred to as “the original part” and “the duplicate part”. They may both belong to the same WitRun or to two different WitRuns.

`dupWitRun`

The WitRun to which the duplicate part belongs.

`dupPartName`

The partName of the duplicate part.

`origWitRun`

The WitRun to which the original part belongs.

`origPartName`

The partName of the original part.

Usage notes

1. Input data attributes (e.g. supplyVol) are copied; attributes computed by WIT (e.g. stockVol) are not copied.
2. The following input attribute is not copied:
   - appData

Error conditions

- Must be preceeded by a call to `witInitialize (origWitRun)`.
- If `witCopyPartData` either returns with a return code ≥ `WitSEVERE_RC` or throws a WitErrorExc, then neither `dupWitRun` nor `origWitRun` may be used in any further API function calls.
- The following global attributes must have the same value for `dupWitRun` and `origWitRun`:
  - nPeriods
  - independentOffsets
- `dupPartName` must identify an existing part in `dupWitRun`.
- `origPartName` must identify an existing part in `origWitRun`.
- The duplicate and original parts must not be the same part in the same WitRun. Thus either `dupWitRun` ≠ `origWitRun`, or `dupPartName` ≠ `origPartName`.

- The partCategory of the duplicate part must match the partCategory of the original part.

Example

This function is similar to `witCopyBomEntryData`. See the example on page 324.

witCopySubsBomEntryData

```
witReturnCode witCopySubsBomEntryData (
   WitRun * const     dupWitRun,
   const char * const dupOperationName,
   const int          dupBomEntryIndex,
   const int          dupSubsBomEntryIndex,
   WitRun * const     origWitRun,
   const char * const origOperationName,
   const int          origBomEntryIndex,
   const int          origSubsBomEntryIndex);
```

Description

This function copies the input data attributes from one substitute BOM entry into another substitute BOM entry. The two substitute BOM entries are referred to as “the original substitute” and “the duplicate substitute”. They may both belong to the same WitRun or to two different WitRuns.

`dupWitRun`

The WitRun to which the duplicate substitute belongs.

`dupOperationName`

The consumingOperationName of the duplicate substitute.

`dupBomEntryIndex`

The bomEntryIndex of the duplicate substitute.

`dupSubsBomEntryIndex`

The subsBomEntryIndex of the duplicate substitute.

`origWitRun`

The WitRun to which the original substitute belongs.

`origOperationName`

The consumingOperationName of the original substitute.

`origBomEntryIndex`

The bomEntryIndex of the original substitute.

`origSubsBomEntryIndex`

The subsBomEntryIndex of the original substitute.

Usage notes

1. Input data attributes (e.g. offset) are copied; attributes computed by WIT (e.g. impactPeriod) are not copied.
2. The following input attribute is not copied:
   - appData

Error conditions

- Must be preceeded by a call to `witInitialize (origWitRun)`.

- If `witCopySubsBomEntryData` either returns with a return code ≥ `WitSEVERE_RC` or throws a WitErrorExc, then neither `dupWitRun` nor `origWitRun` may be used in any further API function calls.
- The following global attributes must have the same value for `dupWitRun` and `origWitRun`:
  - nPeriods
  - independentOffsets
- `dupOperationName`, `dupBomEntryIndex`, and `dupSubsBomEntryIndex` must identify an existing substitute BOM entry in `dupWitRun`.
- `origOperationName`, `origBomEntryIndex`, and `origSubsBomEntryIndex` must identify an existing substitute BOM entry in `origWitRun`.
- The duplicate and original substitutes must not be the same substitute BOM entry in the same WitRun. Thus either `dupWitRun` ≠ `origWitRun`, or `dupOperationName` ≠ `origOperationName`, or `dupBomEntryIndex` ≠ `origBomEntryIndex`, or `dupSubsBomEntryIndex` ≠ `origSubsBomEntryIndex`.

Example

This function is similar to `witCopyBomEntryData`. See the example on page 324.

Object Iteration Functions

witAdvanceObjItr

```
witReturnCode witAdvanceObjItr (
   WitRun * const theWitRun);
```

Description

This function advances the object iteration process. (See “Object Iteration” on page 35.)

`theWitRun`

Identifies the WIT problem to be used by this function.

Example

See “Example of Application Code that Uses Object Iteration Functions” on page 344.

witGetObjItrState

```
witReturnCode witGetObjItrState (
   WitRun * const theWitRun,
   witAttr *      objItrState);
```

Description

This function retrieves the current value of the global attribute “objItrState”.

`theWitRun`

Identifies the WIT problem to be used by this function.

`objItrState`

On return, `(* objItrState)` stores the current value of the global attribute, “objItrState”, which identifies the current state of the object iteration process. The possible values of the objItrState attribute are:

- `WitINACTIVE`
- `WitAT_PART`
- `WitAT_DEMAND`
- `WitAT_OPERATION`
- `WitAT_BOM_ENTRY`
- `WitAT_SUB_ENTRY`
- `WitAT_BOP_ENTRY`

Example

See “Example of Application Code that Uses Object Iteration Functions” on page 344.

witGetObjItrPart

```
witReturnCode witGetObjItrPart (
   WitRun  * const theWitRun,
   char * *        partName);
```

Description

If the object iteration process is currently located at a part, this function retrieves the attribute that identifies that part.

`theWitRun`

Identifies the WIT problem to be used by this function.

`partName`

On return, `(* partName)` stores the partName of the part at which object iteration is currently located.

Usage notes

1. It is the application's responsibility to free the returned string, `(* partName)`.

Error Conditions

- The object iteration process must be located at a part.

Example

See "Example of Application Code that Uses Object Iteration Functions" on page 344.

witGetObjItrDemand

```
witReturnCode witGetObjItrDemand (
   WitRun * const theWitRun,
   char * *       demandedPartName,
   char * *       demandName);
```

Description

If the object iteration process is currently located at a demand, this function retrieves the attributes that identify that demand.

`theWitRun`

Identifies the WIT problem to be used by this function.

`demandedPartName`

On return, `(* demandedPartName)` stores the demandedPartName of the demand at which object iteration is currently located.

`demandName`

On return, `(* demandName)` stores the demandName of the demand at which object iteration is currently located.

Usage notes

1. It is the application’s responsibility to free the returned strings, `(* demandedPartName)` and `(* demandName)`.

Error Conditions

- The object iteration process must be located at a demand.

Example

See “Example of Application Code that Uses Object Iteration Functions” on page 344.

witGetObjItrOperation

```
witReturnCode witGetObjItrOperation (
   WitRun * const theWitRun,
   char * *       operationName);
```

Description

If the object iteration process is currently located at an operation, this function retrieves the attribute that identifies that operation.

`theWitRun`

Identifies the WIT problem to be used by this function.

`operationName`

On return, `(* operationName)` stores the operationName of the operation at which object iteration is currently located.

Usage notes

1. It is the application's responsibility to free the returned string, `(* operationName)`.

Error Conditions

- The object iteration process must be located at an operation.

Example

See "Example of Application Code that Uses Object Iteration Functions" on page 344.

witGetObjItrBomEntry

```
witReturnCode witGetObjItrBomEntry (
   WitRun * const theWitRun,
   char * *       consumingOperationName,
   int *          bomEntryIndex);
```

Description

If the object iteration process is currently located at a BOM entry, this function retrieves the attributes that identify that BOM entry.

`theWitRun`

Identifies the WIT problem to be used by this function.

`consumingOperationName`

On return, `(* consumingOperationName)` stores the consumingOperationName of the BOM entry at which object iteration is currently located.

`bomEntryIndex`

On return, `(* bomEntryIndex)` stores the bomEntryIndex of the BOM entry at which object iteration is currently located.

Usage notes

1. It is the application's responsibility to free the returned string, `(* consumingOperationName)`.

Error Conditions

- The object iteration process must be located at a BOM entry.

Example

See "Example of Application Code that Uses Object Iteration Functions" on page 344.

witGetObjItrSubsBomEntry

```
witReturnCode witGetObjItrSubsBomEntry (
   WitRun * const theWitRun,
   char * *       consumingOperationName,
   int *          bomEntryIndex,
   int *          subsBomEntryIndex);
```

Description

If the object iteration process is currently located at a substitute BOM entry, this function retrieves the attributes that identify that substitute BOM entry.

`theWitRun`

Identifies the WIT problem to be used by this function.

`consumingOperationName`

On return, `(* consumingOperationName)` stores the consumingOperationName of the substitute BOM entry at which object iteration is currently located.

`bomEntryIndex`

On return, `(* bomEntryIndex)` stores the bomEntryIndex of the substitute BOM entry at which object iteration is currently located.

`subsBomEntryIndex`

On return, `(* subsBomEntryIndex)` stores the subsBomEntryIndex of the substitute BOM entry at which object iteration is currently located.

Usage notes

1. It is the application's responsibility to free the returned string, `(* consumingOperationName)`.

Error Conditions

- The object iteration process must be located at a substitute BOM entry.

Example

See “Example of Application Code that Uses Object Iteration Functions” on page 344.

witGetObjItrBopEntry

```
witReturnCode witGetObjItrBopEntry (
   WitRun * const theWitRun,
   char * *       producingOperationName,
   int *          bopEntryIndex);
```

Description

If the object iteration process is currently located at a BOP entry, this function retrieves the attributes that identify that BOP entry.

`theWitRun`

Identifies the WIT problem to be used by this function.

`producingOperationName`

On return, `(* producingOperationName)` stores the producingOperationName of the BOP entry at which object iteration is currently located.

`bopEntryIndex`

On return, `(* bopEntryIndex)` stores the bopEntryIndex of the BOP entry at which object iteration is currently located.

Usage notes

1. It is the application's responsibility to free the returned string, `(* producingOperationName)`.

Error Conditions

- The object iteration process must be located at a BOP entry.

Example

See "Example of Application Code that Uses Object Iteration Functions" on page 344.

witResetObjItr

```
witReturnCode witResetObjItr (
   WitRun * const theWitRun);
```

Description

This function puts the object iteration process into its inactive state. (See "Object Iteration" on page 35.)

`theWitRun`

Identifies the WIT problem to be used by this function.

Example

See "Example of Application Code that Uses Object Iteration Functions" on page 344.

Example of Application Code that Uses Object Iteration Functions

```
/*-------------------------------------------------*/
/* Function expAllowedSubExists (theWitRun)        */
/*                                                 */
/* Returns WitTRUE, iff theWitRun contains at      */
/* least one substitute for which expAllowed is    */
/* TRUE. Object iteration is required to be        */
/* inactive prior to calling this function and is  */
/* restored to an inactive state by this function  */
/* before it returns.                              */
/*-------------------------------------------------*/

witBoolean expAllowedSubExists (WitRun * theWitRun)
   {
   witAttr    objItrState;
   char *     opnName;
   int        bomEntIdx;
   int        subIdx;
   witBoolean expAllowed;

   witGetObjItrState (theWitRun, & objItrState);

   if (objItrState != WitINACTIVE)
      {
      fprintf (stderr,
         "ERROR: Object iteration must be inactive "
         "prior to calling expAllowedSubExists.");

      exit (3);
      }
while (WitTRUE)
   {
   witAdvanceObjItr  (theWitRun);

   witGetObjItrState (theWitRun, & objItrState);

   if (objItrState == WitINACTIVE)
      return WitFALSE;

   if (objItrState == WitAT_SUB_ENTRY)
      {
      witGetObjItrSubsBomEntry (
           theWitRun,
         & opnName,
         & bomEntIdx,
         & subIdx);

      witGetSubsBomEntryExpAllowed (
           theWitRun,
           opnName,
           bomEntIdx,
           subIdx,
         & expAllowed);

      free (opnName);

      if (expAllowed)
         {
         witResetObjItr (theWitRun);

         return WitTRUE;
         }
      }
   }
}
```

Post-Implosion Pegging Functions

witAppendToPipSeq

```
witReturnCode witAppendToPipSeq (
   WitRun * const     theWitRun,
   const char * const partName,
   const char * const demandName,
   const int          shipPeriod,
   const float        incShipVol);
```

Description

Appends a triple to the end of the PIP shipment sequence. See “Post-Implosion Pegging” on page 43.

`theWitRun`

Identifies the WIT problem to be used by this function.

`partName`

The demandedPartName for the demand of the shipment triple to be appended.

`demandName`

The demandName for the demand of the shipment triple to be appended.

`shipPeriod`

The shipment period of the shipment triple to be appended.

`incShipVol`

The incremental shipment volume of the shipment triple to be appended.

Error conditions

- A demand with the specified demandedPartName and demandName must have been previously defined. See also “Error Conditions” on page 222.
- `shipPeriod` must be in the range:

  $0 \leq$ `shipPeriod` $<$ nPeriods
- `incShipVol` must be ≥ 0.0.

Example

```
/*----------------------------------------------------*/
/* This function invokes heuristic implosion and then   */
/* has WIT build a post-implosion pegging using a       */
/* shipment sequence that’s the reverse of the one that */
/* was used by the heuristic.                          */
/*----------------------------------------------------*/
```

```
void buildReverseHeurPip (WitRun * theWitRun)
   {
   int      lenLists;
   char * * partNameList;
   char * * demandNameList;
   int *    shipPerList;
   float *  incShipVolList;
   int      theIdx;

   witSetPipSeqFromHeur (theWitRun, WitTRUE);

   witHeurImplode (theWitRun);

   witGetPipSeq (
        theWitRun,
      & lenLists,
      & partNameList,
      & demandNameList,
      & shipPerList,
      & incShipVolList);

   witClearPipSeq (theWitRun);

   for (theIdx = lenLists - 1; theIdx >= 0; theIdx --)
      witAppendToPipSeq (
         theWitRun,
         partNameList  [theIdx],
         demandNameList[theIdx],
         shipPerList   [theIdx],
         incShipVolList[theIdx]);

   for (theIdx = 0; theIdx < lenLists; theIdx ++)
      {
      free (partNameList[theIdx]);
      free (demandNameList[theIdx]);
      }

   free (partNameList);
   free (demandNameList);
   free (shipPerList);
   free (incShipVolList);

   witBuildPip (theWitRun);
   }
```

witBuildPip

```
witReturnCode witBuildPip (
   WitRun * const theWitRun);
```

Description

Builds a post-implosion pegging for the current implosion solution. See “Post-Implosion Pegging” on page 43.

theWitRun

Identifies the WIT problem to be used by this function.

Error conditions

- `theWitRun` must be in a postprocessed state.

Example

See “witAppendToPipSeq” on page 346.

witClearPipSeq

```
witReturnCode witClearPipSeq (
   WitRun * const theWitRun);
```

Description

Clears (i.e., empties) the PIP shipment sequence. See “Post-Implosion Pegging” on page 43.

theWitRun

Identifies the WIT problem to be used by this function.

Example

See “witAppendToPipSeq” on page 346.

witGetDemandCoExecVolPip

```
witReturnCode witGetDemandCoExecVolPip (
   WitRun * const     theWitRun,
   const char * const demandedPartName,
   const char * const demandName,
   const int          shipPeriod,
   int *              lenLists,
   char * * *         operationNameList,
   int * *            bopEntryIndexList,
   int * *            execPeriodList,
   float * *          peggedCoExecVolList);
```

Description

Retrieves the post-implosion coExecVol pegging associated with a specific demand in a specific shipment period. See “Post-Implosion Pegging” on page 43. The application program specifies a demand and shipment period with the arguments `demandedPartName`, `demandName`, and `shipPeriod`. On return, for `0 ≤ i < (*lenLists)`, `(*operationNameList)[i]` and `(*bopEntryIndexList)[i]` identify a BOP entry, `(*execPeriodList)[i]` identifies an execution period, and `(*peggedCoExecVolList)[i]` is a pegged coExecVol. The pegged coExecVol is the portion of the execVol of the operation in the execution period that’s pegged to the demand in `shipPeriod` and whose pegging is specifically associated with the BOP entry.

`theWitRun`

Identifies the WIT problem to be used by this function.

`demandedPartName`

The demandedPartName for the demand whose coExecVol pegging is to be returned (i.e., the partName of the demanded part).

`demandName`

The demandName for the demand whose coExecVol pegging is to be returned.

`shipPeriod`

The shipment period whose coExecVol pegging is to be returned.

`lenLists`

On return, `(*lenLists)` is the number of pegging triples retrieved.

`operationNameList`

On return, for `0 ≤ i < (*lenLists)`, `(*operationNameList)[i]` is the producingOperationName of the BOP entry for pegging triple #`i`.

`bopEntryIndexList`

On return, for `0 ≤ i < (*lenLists)`, `(*bopEntryIndexList)[i]` is the bopEntryIndex of the BOP entry for pegging triple #`i`.

`execPeriodList`

On return, for `0 ≤ i < (*lenLists)`, `(*execPeriodList)[i]` is the execution period for pegging triple #`i`.

`peggedCoExecVolList`

On return, for `0 ≤ i < (*lenLists)`, `(*peggedCoExecVolList)[i]` is the pegged coExecVol for pegging triple #`i`.

Usage notes

- It is the responsibility of the application to free the returned vectors.

Error conditions

- A demand with the specified demandedPartName and demandName must have been previously defined. See also "Error Conditions" on page 222.
- `shipPeriod` must be in the range:

  0 ≤ `shipPeriod` < nPeriods
- The pipExists global attribute must be TRUE.

Example

`witGetDemandCoExecVolPip` works similarly to `witGetDemandSupplyVolPip`.

See "witGetDemandSupplyVolPip" on page 356 for an example of its use.

witGetDemandExecVolPip

```
witReturnCode witGetDemandExecVolPip (
   WitRun * const     theWitRun,
   const char * const demandedPartName,
   const char * const demandName,
   const int          shipPeriod,
   int *              lenLists,
   char * * *         operationNameList,
   int * *            execPeriodList,
   float * *          peggedExecVolList);
```

Description

Retrieves the post-implosion execVol pegging associated with a specific demand in a specific shipment period. See “Post-Implosion Pegging” on page 43. The application program specifies a demand and shipment period with the arguments `demandedPartName`, `demandName`, and `shipPeriod`. On return, for `0 ≤ i < (*lenLists)`, `(*operationNameList)[i]` identifies an operation, `(*execPeriodList)[i]` identifies an execution period, and `(*peggedExecVolList)[i]` is a pegged execVol. The pegged execVol is the portion of the execVol of the operation in the execution period that’s pegged to the demand in `shipPeriod`.

`theWitRun`

Identifies the WIT problem to be used by this function.

`demandedPartName`

The demandedPartName for the demand whose execVol pegging is to be returned (i.e., the partName of the demanded part).

`demandName`

The demandName for the demand whose execVol pegging is to be returned.

`shipPeriod`

The shipment period whose execVol pegging is to be returned.

`lenLists`

On return, `(*lenLists)` is the number of pegging triples retrieved.

`operationNameList`

On return, for `0 ≤ i < (*lenLists)`,`(*operationNameList)[i]` is the operationName of the operation for pegging triple #`i`.

`execPeriodList`

On return, for `0 ≤ i < (*lenLists)`, `(*execPeriodList)[i]` is the execution period for pegging triple #`i`.

`peggedExecVolList`

On return, for `0 ≤ i < (*lenLists)`, `(*peggedExecVolList)[i]` is the pegged execVol for pegging triple #`i`.

Usage notes

- It is the responsibility of the application to free the returned vectors.

Error conditions

- A demand with the specified demandedPartName and demandName must have been previously defined. See also “Error Conditions” on page 222.
- `shipPeriod` must be in the range:

  0 ≤ `shipPeriod` < nPeriods
- The pipExists global attribute must be TRUE.

Example

`witGetDemandExecVolPip` works similarly to `witGetDemandSupplyVolPip`.

See “witGetDemandSupplyVolPip” on page 356 for an example of its use.

witGetDemandSubVolPip

```
witReturnCode witGetDemandSubVolPip (
   WitRun * const     theWitRun,
   const char * const demandedPartName,
   const char * const demandName,
   const int          shipPeriod,
   int *              lenLists,
   char * * *         operationNameList,
   int * *            bomEntryIndexList,
   int * *            subsBomEntryIndexList,
   int * *            execPeriodList,
   float * *          peggedSubVolList);
```

Description

Retrieves the post-implosion subVol pegging associated with a specific demand in a specific shipment period. See “Post-Implosion Pegging” on page 43. The application program specifies a demand and shipment period with the arguments `demandedPartName`, `demandName`, and `shipPeriod`. On return, for `0 ≤ i < (*lenLists)`, `(*operationNameList)[i]`, `(*bomEntryIndexList)[i]`, and`(*subsBomEntryIndexList)[i]` identify a substitute BOM entry, `(*execPeriodList)[i]` identifies an execution period, and `(*peggedSubVolList)[i]` is a pegged subVol. The pegged subVol is the portion of the subVol of the substitute in the execution period that’s pegged to the demand in `shipPeriod`.

`theWitRun`

Identifies the WIT problem to be used by this function.

`demandedPartName`

The demandedPartName for the demand whose subVol pegging is to be returned (i.e., the partName of the demanded part).

`demandName`

The demandName for the demand whose subVol pegging is to be returned.

`shipPeriod`

The shipment period whose subVol pegging is to be returned.

`lenLists`

On return, `(*lenLists)` is the number of pegging triples retrieved.

`operationNameList`

On return, for `0 ≤ i < (*lenLists)`, `(*operationNameList)[i]` is the consumingOperationName of the substitute BOM entry for pegging triple #`i`.

`bomEntryIndexList`

On return, for `0 ≤ i < (*lenLists)`, `(*bomEntryIndexList)[i]` is the bomEntryIndex of the substitute BOM entry for pegging triple #`i`.

`subsBomEntryIndexList`

On return, for `0 ≤ i < (*lenLists)`, `(*subsBomEntryIndexList)[i]` is the subsBomEntryIndex of the substitute BOM entry for pegging triple #`i`.

`execPeriodList`

On return, for `0 ≤ i < (*lenLists)`, `(*execPeriodList)[i]` is the execution period for pegging triple #`i`.

`peggedSubVolList`

On return, for `0 ≤ i < (*lenLists)`, `(*peggedSubVolList)[i]` is the pegged subVol for pegging triple #`i`.

Usage notes

- It is the responsibility of the application to free the returned vectors.

Error conditions

- A demand with the specified demandedPartName and demandName must have been previously defined. See also “Error Conditions” on page 222.
- `shipPeriod` must be in the range:

  0 ≤ `shipPeriod` < nPeriods
- The pipExists global attribute must be TRUE.

Example

`witGetDemandSubVolPip` works similarly to `witGetDemandSupplyVolPip`.

See “witGetDemandSupplyVolPip” on page 356 for an example of its use.

witGetDemandSupplyVolPip

```
witReturnCode witGetDemandSupplyVolPip (
   WitRun * const     theWitRun,
   const char * const demandedPartName,
   const char * const demandName,
   const int          shipPeriod,
   int *              lenLists,
   char * * *         partNameList,
   int * *            periodList,
   float * *          peggedSupplyVolList);
```

Description

Retrieves the post-implosion supplyVol pegging associated with a specific demand in a specific shipment period. See “Post-Implosion Pegging” on page 43. The application program specifies a demand and shipment period with the arguments `demandedPartName`, `demandName`, and `shipPeriod`. On return, for `0 ≤ i < (*lenLists)`, `(*partNameList)[i]` identifies a part, `(*periodList)[i]` identifies a period, and `(*peggedSupplyVolList)[i]` is a pegged supplyVol. The pegged supplyVol is the portion of the supplyVol of the part in the period that’s pegged to the demand in `shipPeriod`.

`theWitRun`

Identifies the WIT problem to be used by this function.

`demandedPartName`

The demandedPartName for the demand whose supplyVol pegging is to be returned (i.e., the partName of the demanded part).

`demandName`

The demandName for the demand whose supplyVol pegging is to be returned.

`shipPeriod`

The shipment period whose supplyVol pegging is to be returned.

`lenLists`

On return, `(*lenLists)` is the number of pegging triples retrieved.

`partNameList`

On return, for `0 ≤ i < (*lenLists)`,`(*partNameList)[i]` is the partName of the part for pegging triple #`i`.

`periodList`

On return, for `0 ≤ i < (*lenLists), (*periodList)[i]` is the period for pegging triple #`i`.

`peggedSupplyVolList`

On return, for `0 ≤ i < (*lenLists)`, `(*peggedSupplyVolList)[i]` is the pegged supplyVol for pegging triple `#i`.

Usage notes

- It is the responsibility of the application to free the returned vectors.

Error conditions

- A demand with the specified demandedPartName and demandName must have been previously defined. See also “Error Conditions” on page 222.
- `shipPeriod` must be in the range:

  0 ≤ `shipPeriod` < nPeriods
- The pipExists global attribute must be TRUE.

Example

```
void printDemandSupplyVolPip (
      WitRun *      theWitRun,
      const char * partName,
      const char * demandName,
      int          shipPer)
   {
   int      lenLists;
   char * * partNameList;
   int *    periodList;
   float *  peggedVolList;
   int      theIdx;

   witGetDemandSupplyVolPip (
      theWitRun,
      partName,
      demandName,
      shipPer,
    & lenLists,
    & partNameList,
    & periodList,
    & peggedVolList);

   printf (
      "Post Implosion Demand SupplyVol Pegging:\n"
      "   Demand %s, Part %s, Period %d:\n\n",
      demandName,
      partName,
      shipPer);

   for (theIdx = 0; theIdx < lenLists; theIdx ++)
      printf (
         "      Part %s, Period %d, SupplyVol: %.0f\n",
         partNameList [theIdx],
         periodList   [theIdx],
         peggedVolList[theIdx]);

   for (theIdx = 0; theIdx < lenLists; theIdx ++)
      free (partNameList[theIdx]);

   free (partNameList);
   free (periodList);
   free (peggedVolList);
   }
```

The function call:

```
printDemandSupplyVolPip (theWitRun, "D", "I", 0);
```

might produce the following output:

```
Post Implosion Demand SupplyVol Pegging:
   Demand I, Part D, Period 3:

      Part B, Period 2, SupplyVol: 30
      Part A, Period 0, SupplyVol: 50
```

witGetDemand{Part Attribute}Pip

The following functions retrieve a post-implosion pegging for consVol, prodVol and sideVol. These functions work very similarly to witGetDemandSupplyVolPip. See “witGetDemandSupplyVolPip” on page 356 for more information.

witGetDemandConsVolPip

```
witReturnCode witGetDemandConsVolPip (
   WitRun * const     theWitRun,
   const char * const demandedPartName,
   const char * const demandName,
   const int          shipPeriod,
   int *              lenLists,
   char * * *         partNameList,
   int * *            periodList,
   float * *          peggedConsVolList);
```

witGetDemandProdVolPip

```
witReturnCode witGetDemandProdVolPip (
   WitRun * const     theWitRun,
   const char * const demandedPartName,
   const char * const demandName,
   const int          shipPeriod,
   int *              lenLists,
   char * * *         partNameList,
   int * *            periodList,
   float * *          peggedProdVolList);
```

witGetDemandSideVolPip

```
witReturnCode witGetDemandSideVolPip (
   WitRun * const     theWitRun,
   const char * const demandedPartName,
   const char * const demandName,
   const int          shipPeriod,
   int *              lenLists,
   char * * *         partNameList,
   int * *            periodList,
   float * *          peggedSideVolList);
```

witGetOperationCoExecVolPip

```
witReturnCode witGetOperationCoExecVolPip (
   WitRun * const     theWitRun,
   const char * const operationName,
   const int          execPeriod,
   int *              lenLists,
   char * * *         operationNameList,
   int * *            bopEntryIndexList,
   int * *            execPeriodList,
   float * *          peggedCoExecVolList);
```

Description

Retrieves the post-implosion coExecVol pegging associated with a specific operation in a specific execution period. See “PIP to Operations” on page 50. The application program specifies a pegging operation and execution period with the arguments `operationName` and `execPeriod`. On return, for `0 ≤ i < (*lenLists)`, `(*operationNameList)[i]` and `(*bopEntryIndexList)[i]` identify a BOP entry, `(*execPeriodList)[i]` identifies a period, and `(*peggedCoExecVolList)[i]` is a pegged coExecVol. The pegged coExecVol is the portion of the execVol of the operation in the period that’s pegged to the pegging operation in `execPeriod` and whose pegging is specifically associated with the BOP entry.

`theWitRun`

Identifies the WIT problem to be used by this function.

`operationName`

The operationName for the pegging operation whose coExecVol pegging is to be returned.

`execPeriod`

The execution period whose coExecVol pegging is to be returned.

`lenLists`

On return, `(*lenLists)` is the number of pegging triples retrieved.

`operationNameList`

On return, for `0 ≤ i < (*lenLists)`, `(*operationNameList)[i]` is the producingOperationName of the BOP entry for pegging triple #`i`.

`bopEntryIndexList`

On return, for `0 ≤ i < (*lenLists)`, `(*bopEntryIndexList)[i]` is the bopEntryIndex of the BOP entry for pegging triple #`i`.

`execPeriodList`

On return, for `0 ≤ i < (*lenLists)`, `(*execPeriodList)[i]` is the period for pegging triple #`i`.

`peggedCoExecVolList`

On return, for `0 ≤ i < (*lenLists)`, `(*peggedCoExecVolList)[i]` is the pegged coExecVol for pegging triple #`i`.

Usage notes

- It is the responsibility of the application to free the returned vectors.

Error conditions

- An operation with the specified operationName must have been previously defined. See also “Error Conditions” on page 222.
- The pipEnabled attribute must be TRUE for the operation with the specified operationName.
- `execPeriod` must be in the range:

  0 ≤ `execPeriod` < nPeriods
- The pipExists global attribute must be TRUE.

Example

`witGetOperationCoExecVolPip` works similarly to `witGetOperationSupplyVolPip`.

See “witGetOperationSupplyVolPip” on page 367 for an example of its use.

witGetOperationExecVolPip

```
witReturnCode witGetOperationExecVolPip (
   WitRun * const     theWitRun,
   const char * const operationName,
   const int          execPeriod,
   int *              lenLists,
   char * * *         operationNameList,
   int * *            execPeriodList,
   float * *          peggedExecVolList);
```

Description

Retrieves the post-implosion execVol pegging associated with a specific operation in a specific execution period. See “PIP to Operations” on page 50. The application program specifies a pegging operation and execution period with the arguments `operationName` and `execPeriod`. On return, for `0 ≤ i < (*lenLists)`, `(*operationNameList)[i]` identifies an operation, `(*execPeriodList)[i]` identifies a period, and `(*peggedExecVolList)[i]` is a pegged execVol. The pegged execVol is the portion of the execVol of the operation in the period that’s pegged to the pegging operation in `execPeriod`.

`theWitRun`

Identifies the WIT problem to be used by this function.

`operationName`

The operationName for the pegging operation whose execVol pegging is to be returned.

`execPeriod`

The execution period whose execVol pegging is to be returned.

`lenLists`

On return, `(*lenLists)` is the number of pegging triples retrieved.

`operationNameList`

On return, for `0 ≤ i < (*lenLists),(*operationNameList)[i]` is the operationName of the operation for pegging triple #`i`.

`execPeriodList`

On return, for `0 ≤ i < (*lenLists), (*execPeriodList)[i]` is the period for pegging triple #`i`.

`peggedExecVolList`

On return, for `0 ≤ i < (*lenLists), (*peggedExecVolList)[i]` is the pegged execVol for pegging triple #`i`.

Usage notes

- It is the responsibility of the application to free the returned vectors.

Error conditions

- An operation with the specified operationName must have been previously defined. See also “Error Conditions” on page 222.
- The pipEnabled attribute must be TRUE for the operation with the specified operationName.
- `execPeriod` must be in the range:

  0 ≤ `execPeriod` < nPeriods
- The pipExists global attribute must be TRUE.

Example

`witGetOperationExecVolPip` works similarly to `witGetOperationSupplyVolPip`.

See “witGetOperationSupplyVolPip” on page 367 for an example of its use.

witGetOperationSubVolPip

```
witReturnCode witGetOperationSubVolPip (
   WitRun * const     theWitRun,
   const char * const operationName,
   const int          execPeriod,
   int *              lenLists,
   char * * *         operationNameList,
   int * *            bomEntryIndexList,
   int * *            subsBomEntryIndexList,
   int * *            execPeriodList,
   float * *          peggedSubVolList);
```

Description

Retrieves the post-implosion subVol pegging associated with a specific operation in a specific execution period. See “PIP to Operations” on page 50. The application program specifies a pegging operation and execution period with the arguments `operationName` and `execPeriod`. On return, for `0 ≤ i < (*lenLists)`, `(*operationNameList)[i]`, `(*bomEntryIndexList)[i]`, and `(*subsBomEntryIndexList)[i]` identify a substitute BOM entry, `(*execPeriodList)[i]` identifies a period, and `(*peggedSubVolList)[i]` is a pegged subVol. The pegged subVol is the portion of the subVol of the substitute in the period that’s pegged to the pegging operation in `execPeriod`.

`theWitRun`

Identifies the WIT problem to be used by this function.

`operationName`

The operationName for the pegging operation whose subVol pegging is to be returned.

`execPeriod`

The execution period whose subVol pegging is to be returned.

`lenLists`

On return, `(*lenLists)` is the number of pegging triples retrieved.

`operationNameList`

On return, for `0 ≤ i < (*lenLists)`, `(*operationNameList)[i]` is the consumingOperationName of the substitute BOM entry for pegging triple #`i`.

`bomEntryIndexList`

On return, for `0 ≤ i < (*lenLists)`, `(*bomEntryIndexList)[i]` is the bomEntryIndex of the substitute BOM entry for pegging triple #`i`.

`subsBomEntryIndexList`

On return, for `0 ≤ i < (*lenLists)`, `(*subsBomEntryIndexList)[i]` is the subsBomEntryIndex of the substitute BOM entry for pegging triple #`i`.

`execPeriodList`

On return, for `0 ≤ i < (*lenLists)`, `(*execPeriodList)[i]` is the period for pegging triple #`i`.

`peggedSubVolList`

On return, for `0 ≤ i < (*lenLists)`, `(*peggedSubVolList)[i]` is the pegged subVol for pegging triple #`i`.

Usage notes

- It is the responsibility of the application to free the returned vectors.

Error conditions

- An operation with the specified operationName must have been previously defined. See also “Error Conditions” on page 222.
- The pipEnabled attribute must be TRUE for the operation with the specified operationName.
- `execPeriod` must be in the range:

  0 ≤ `execPeriod` < nPeriods
- The pipExists global attribute must be TRUE.

Example

`witGetOperationSubVolPip` works similarly to `witGetOperationSupplyVolPip`.

See “witGetOperationSupplyVolPip” on page 367 for an example of its use.

witGetOperationSupplyVolPip

```
witReturnCode witGetOperationSupplyVolPip (
   WitRun * const      theWitRun,
   const char * const  operationName,
   const int           execPeriod,
   int *               lenLists,
   char * * *          partNameList,
   int * *             periodList,
   float * *           peggedSupplyVolList);
```

Description

Retrieves the post-implosion supplyVol pegging associated with a specific operation in a specific execution period. See “PIP to Operations” on page 50. The application program specifies a pegging operation and execution period with the arguments `operationName` and `execPeriod`. On return, for `0 ≤ i < (*lenLists)`, `(*partNameList)[i]` identifies a part, `(*periodList)[i]` identifies a period, and `(*peggedSupplyVolList)[i]` is a pegged supplyVol. The pegged supplyVol is the portion of the supplyVol of the part in the period that’s pegged to the pegging operation in `execPeriod`.

`theWitRun`

Identifies the WIT problem to be used by this function.

`operationName`

The operationName for the pegging operation whose supplyVol pegging is to be returned.

`execPeriod`

The execution period whose supplyVol pegging is to be returned.

`lenLists`

On return, `(*lenLists)` is the number of pegging triples retrieved.

`partNameList`

On return, for `0 ≤ i < (*lenLists)`,`(*partNameList)[i]` is the partName of the part for pegging triple #`i`.

`periodList`

On return, for `0 ≤ i < (*lenLists)`, `(*periodList)[i]` is the period for pegging triple #`i`.

`peggedSupplyVolList`

On return, for `0 ≤ i < (*lenLists)`, `(*peggedSupplyVolList)[i]` is the pegged supplyVol for pegging triple #`i`.

Usage notes

- It is the responsibility of the application to free the returned vectors.

Error conditions

- An operation with the specified operationName must have been previously defined. See also “Error Conditions” on page 222.
- The pipEnabled attribute must be TRUE for the operation with the specified operationName.
- `execPeriod` must be in the range:

  $0 \leq$ `execPeriod` < nPeriods
- The pipExists global attribute must be TRUE.

Example

```
void printOperationSupplyVolPip (
      WitRun *     theWitRun,
      const char * operationName,
      int          execPer)
   {
   int      lenLists;
   char * * partNameList;
   int *    periodList;
   float *  peggedVolList;
   int      theIdx;

   witGetOperationSupplyVolPip (
      theWitRun,
      operationName,
      execPer,
    & lenLists,
    & partNameList,
    & periodList,
    & peggedVolList);

   printf (
      "Post Implosion Operation SupplyVol Pegging:\n"
      "   Operation %s, Period %d:\n\n",
      operationName,
      execPer);

   for (theIdx = 0; theIdx < lenLists; theIdx ++)
      {
      printf (
         "      Part %s, Period %d, SupplyVol: %.0f\n",
         partNameList [theIdx],
         periodList   [theIdx],
         peggedVolList[theIdx]);
      }

   for (theIdx = 0; theIdx < lenLists; theIdx ++)
      free (partNameList[theIdx]);

   free (partNameList);
   free (periodList);
   free (peggedVolList);
   }
```

The function call:

```
printOperationSupplyVolPip (theWitRun, "H", 3);
```

might produce the following output:

```
Post Implosion Operation SupplyVol Pegging:
   Operation H, Period 3:

      Part B, Period 2, SupplyVol: 30
      Part A, Period 0, SupplyVol: 50
```

witGetOperation{Part Attribute}Pip

The following functions retrieve a post-implosion pegging for consVol, prodVol and sideVol. These functions work very similarly to witGetOperationSupplyVolPip. See “witGetOperationSupplyVolPip” on page 367 for more information.

witGetOperationConsVolPip

```
witReturnCode witGetOperationConsVolPip (
   WitRun * const     theWitRun,
   const char * const operationName,
   const int          execPeriod,
   int *              lenLists,
   char * * *         partNameList,
   int * *            periodList,
   float * *          peggedConsVolList);
```

witGetOperationProdVolPip

```
witReturnCode witGetOperationProdVolPip (
   WitRun * const     theWitRun,
   const char * const operationName,
   const int          execPeriod,
   int *              lenLists,
   char * * *         partNameList,
   int * *            periodList,
   float * *          peggedProdVolList);
```

witGetOperationSideVolPip

```
witReturnCode witGetOperationSideVolPip (
   WitRun * const     theWitRun,
   const char * const operationName,
   const int          execPeriod,
   int *              lenLists,
   char * * *         partNameList,
   int * *            periodList,
   float * *          peggedSideVolList);
```

witGetPipSeq

```
witReturnCode witGetPipSeq (
   WitRun * const theWitRun,
   int *          lenLists,
   char * * *     partNameList,
   char * * *     demandNameList,
   int * *        shipPerList,
   float * *      incShipVolList);
```

Description

Retrieves the PIP shipment sequence. This is an ordered list of shipment triples (demand, shipment period, incShipVol) that is used as an input to post-implosion pegging. See “Post-Implosion Pegging” on page 43.

theWitRun

Identifies the WIT problem to be used by this function.

`lenLists`

On return, (`*lenLists`) is the number of shipment triples retrieved.

`partNameList`

On return, for `0 ≤ i < (*lenLists)`,`(*partNameList)[i]` is the demandedPartName of the demand for shipment triple #`i`.

`demandNameList`

On return, for `0 ≤ i < (*lenLists)`,`(*demandNameList)[i]` is the demandName of the demand for shipment triple #`i`.

`shipPerList`

On return, for `0 ≤ i < (*lenLists)`, `(*shipPerList)[i]` is the shipment period for pegging triple #`i`.

`incShipVolList`

On return, for `0 ≤ i < (*lenLists)`, `(*incShipVolList)[i]` is the incShipVol for pegging triple #`i`.

Usage notes

- It is the responsibility of the application to free the returned vectors.

Example

See “witAppendToPipSeq” on page 346.

CPLEX Parameter Specification Functions

witAddDblCplexParSpec

```
witReturnCode witAddDblCplexParSpec (
   WitRun * const      theWitRun,
   const char * const  theName,
   const float         theValue);
```

Description

Creates a new CPLEX parameter specification of type double. Deletes any existing CPLEX parameter specification in theWitRun whose name matches theName. See “CPLEX Parameter Specifications” on page 66.

`theWitRun`

Identifies the WIT problem to be used by this function.

`theName`

The name of the CPLEX parameter specification to be created.

`theValue`

The value of the CPLEX parameter specification to be created.

Error Conditions

- theName must not be “NO_PARAM” (detected immediately).
- theName must match (when prefixed with “CPX_PARAM_”) the name of a CPLEX parameter as defined for the CPLEX callable library (detected during optimizing or stochastic implosion).
- The named CPLEX parameter must be of type double (detected during optimizing or stochastic implosion).

Usage notes

- As always, this function has a double precision variant. `See “Double Precision Functions” on page 389.`

Example

```
witAddDblCplexParSpec (
   theWitRun,
   "TILIM",
   60.0);
```

This code will cause WIT to create a CPLEX parameter specification of type double with the name “TILIM” and value 60.0. As a result, during optimizing or stochastic implosion with CPLEX, WIT will call CPLEX to set the value of the double CPLEX parameter CPX_PARAM_TILIM to 60.0.

witAddIntCplexParSpec

```
witReturnCode witAddIntCplexParSpec (
   WitRun * const     theWitRun,
   const char * const theName,
   const int          theValue);
```

Description

Creates a new CPLEX parameter specification of type int. Deletes any existing CPLEX parameter specification in theWitRun whose name matches theName. See “CPLEX Parameter Specifications” on page 66.

`theWitRun`

Identifies the WIT problem to be used by this function.

`theName`

The name of the CPLEX parameter specification to be created.

`theValue`

The value of the CPLEX parameter specification to be created.

Error Conditions

- theName must not be “NO_PARAM” (detected immediately).
- theName must match (when prefixed with “CPX_PARAM_”) the name of a CPLEX parameter as defined for the CPLEX callable library (detected during optimizing or stochastic implosion).
- The named CPLEX parameter must be of type int or long (detected during optimizing or stochastic implosion).

Example

```
witAddIntCplexParSpec (
   theWitRun,
   "LPMETHOD",
   1);
```

This code will cause WIT to create a CPLEX parameter specification of type int with the name “LPMETHOD” and value 1. As a result, during optimizing or stochastic implosion with CPLEX, WIT will call CPLEX to set the value of the int CPLEX parameter CPX_PARAM_LPMETHOD to 1.

witClearCplexParSpecs

```
witReturnCode witClearCplexParSpecs (
   WitRun * const theWitRun);
```

Description

This function deletes all existing CPLEX parameter specifications. See "CPLEX Parameter Specifications" on page 66.

`theWitRun`

Identifies the WIT problem to be used by this function.

Example

```
witClearCplexParSpecs (theWitRun);
```

This code deletes all existing CPLEX parameter specifications in `theWitRun`.

witGetDblCplexParSpec

```
witReturnCode witGetDblCplexParSpec (
   WitRun * const     theWitRun,
   const char * const theName,
   witBoolean *       dblSpecExists,
   float *            theValue);
```

Description

This function retrieves a boolean indicating whether or not a CPLEX parameter specification of type double with a specified name currently exists and if it does, this function retrieves its value. See “CPLEX Parameter Specifications” on page 66.

`theWitRun`

Identifies the WIT problem to be used by this function.

`theName`

The specified name.

`dblSpecExists`

On return, `(* dblSpecExists)` is WitTRUE, if a CPLEX parameter specification of type double whose name matches `theName` currently exists. Otherwise `(* dblSpecExists)` is WitFALSE on return.

`theValue`

On return, if `(* dblSpecExists)` is WitTRUE, then`(* theValue)` is the value of the CPLEX parameter specification of type double whose name matches `theName`. If `(* dblSpecExists)` is WitFALSE on return, then`(* theValue)` is left unaltered.

Usage notes

- As always, this function has a double precision variant. `See “Double Precision Functions” on page 389.`

Example

```
witBoolean intSpecExists;
float      theValue;

witGetDblCplexParSpec (
   theWitRun,
   "TILIM",
 & dblSpecExists,
 & theValue);

if (dblSpecExists)
   printf ("TILIM = %g\n", theValue);
else
   printf ("TILIM is unspecified.\n");
```

witGetIntCplexParSpec

```
witReturnCode witGetIntCplexParSpec (
   WitRun * const     theWitRun,
   const char * const theName,
   witBoolean *       intSpecExists,
   int *              theValue);
```

Description

This function retrieves a boolean indicating whether or not a CPLEX parameter specification of type int with a specified name currently exists and if it does, this function retrieves its value. See “CPLEX Parameter Specifications” on page 66.

`theWitRun`

Identifies the WIT problem to be used by this function.

`theName`

The specified name.

`intSpecExists`

On return, `(* intSpecExists)` is WitTRUE, if a CPLEX parameter specification of type int whose name matches `theName` currently exists. Otherwise `(* intSpecExists)` is WitFALSE on return.

`theValue`

On return, if `(* intSpecExists)` is WitTRUE, then`(* theValue)` is the value of the CPLEX parameter specification of type int whose name matches `theName`. If `(* intSpecExists)` is WitFALSE on return, then`(* theValue)` is left unaltered.

Example

```
witBoolean intSpecExists;
int        theValue;

witGetIntCplexParSpec (
   theWitRun,
   "LPMETHOD",
 & intSpecExists,
 & theValue);

if (intSpecExists)
   printf ("LPMETHOD = %d\n", theValue);
else
   printf ("LPMETHOD is unspecified.\n");
```

Message Control Functions

witGetMesgAttribute

```
witReturnCode witGetMesgAttribute
(    WitRun * const theWitRun,
     Type value );
```

or

```
witReturnCode witGetMesgAttribute
(    WitRun * const theWitRun,
     const int messageNumber,
     Type value );
```

`witGet`*`MesgAttribute`* represents a group of functions for retrieving the value of message attributes. The first form given above is for attributes associated with messages in general; the second form is for attributes associated with individual messages. For more information on message attributes, see “API Message Attributes” on page 156.

The `witGet`*`MesgAttribute`* functions are:

```
witReturnCode witGetMesgFile
(    WitRun * const theWitRun,
     FILE  * * mesgFile );
witReturnCode witGetMesgFileAccessMode
(    WitRun  * const theWitRun,
     char * * mesgFileAccessMode );
 witReturnCode witGetMesgFileName
(    WitRun  * const theWitRun,
     char * * mesgFileName );
witReturnCode witGetMesgPrintNumber
(    WitRun  * const theWitRun,
     const int messageNumber,
     witBoolean * mesgPrintNumber );
witReturnCode witGetMesgStopRunning
(    WitRun  * const theWitRun,
     const int messageNumber,
     witBoolean * mesgStopRunning );
witReturnCode witGetMesgThrowErrorExc
(    WitRun  * const theWitRun,
     const int messageNumber,
     witBoolean * mesgThrowErrorExc );
witReturnCode witGetMesgTimesPrint
(    WitRun  * const theWitRun,
      const int messageNumber,
      int * mesgTimesPrint );
```

Description

`theWitRun`

Identifies the WIT problem to be used by this function.

`messageNumber`

When present, the attribute for this message will be retrieved.

`value`

Location where the attribute value is to be returned.

Usage Notes

1. Concerning `witGetMesgFileAccessMode` and `witGetMesgFileName` — It is the responsibility of the application to free the returned vector.

Example

```
FILE * mesgFile;
int    mesgTimesPrint;

witGetMesgFile (theWitRun, &mesgFile );

fprintf (mesgFile,
     "Application mesg written to WIT mesg file\n");

witGetMesgTimesPrint (theWitRun, 98, & mesgTimesPrint);

printf (
    "Message #98 will be printed at most %d times.\n",
    mesgTimesPrint);
```

witSetMesgFileAccessMode

```
witReturnCode witSetMesgFileAccessMode
(    WitRun  * const theWitRun,
     const witBoolean quiet,
     const char * const mesgFileAccessMode );
```

Description

Sets the access mode used in the C `fopen` function when opening the WIT message file.

`theWitRun`

Identifies the WIT problem to be used by this function.

`quiet`

Indicates if the display of informational messages is to be suppressed for this function invocation.

`mesgFileAccessMode`

The new access mode to be used in the future when opening the WIT message file.

Usage Notes

1. This access mode does not affect the LP Solver log file.
2. Calls to `witInitialize` does not change the `mesgFileAccessMode`.
3. See the ANSI C `fopen` function for a description of access modes.

Exceptions to General Error Conditions

- Need not be preceeded by a call to `witInitialize`.

Example

```
witReturnCode rc;
rc = witSetMesgFileAccessMode(theWitRun, WitFALSE, "a" );
```

The access mode for opening the WIT message file is changed to `"a"`.

witSetMesgFileName

```
 witReturnCode witSetMesgFileName
(    WitRun  * const theWitRun,
     const witBoolean quiet,
     const char  * const mesgFileName );
```

Description

Defines the name of the file WIT will use for writing messages.

`theWitRun`

Identifies the WIT problem to be used by this function.

`quiet`

Indicates if the display of informational messages is to be suppressed for this function invocation.

`mesgFileName`

The name of the file where WIT will start writing messages. If either `WitSTDOUT` or `"stdout"` is specified, messages will be written to `stdout`.

Usage Notes

1. The format of `mesgFileName` depends on the particular platform.
2. The currently opened message file is closed if it is not stdout.
3. The file is opened with the ANSI C function `fopen` using the mode `mesgFileAccessMode`.
4. Calls to `witInitialize` do not change the `mesgFileName`.
5. If the `mesgFileName` is not `WitSTDOUT` or `"stdout"`, then the `mesgFileName` should be unique among all WitRuns.

Error Conditions

- Errors reported by `fopen`.

Exceptions to General Error Conditions

- Need not be preceeded by a call to `witInitialize`.

Example

```
witReturnCode rc;
rc = witSetMesgFileName ( theWitRun, witTRUE,
             "/tmp/wit.out" );
```

The currently open message file is closed. The file `/tmp/wit.out` is opened. WIT messages will now be written to this file.

witSetMesgPrintNumber

```
witReturnCode witSetMesgPrintNumber
(    WitRun  * const theWitRun,
     const witBoolean quiet,
     const int messageNumber,
     const witBoolean mesgPrintNumber );
```

Description

Turns on or off the printing of message numbers.

`theWitRun`

Identifies the WIT problem to be used by this function.

`quiet`

Indicates if the display of informational messages is to be suppressed for this function invocation.

`messageNumber`

The `mesgPrintNumber` attribute for this message will be changed.

`WitINFORMATIONAL_MESSAGES, WitWARNING_MESSAGES, or WitSEVERE_MESSAGES` can be specified to change all informational, warning, or severe messages, respectively.

`mesgPrintNumber`

`WitTRUE` indicates the message will be printed with its message number. `WitFALSE` indicates the message will be printed without a message number.

Usage Notes

1. Calls to `witInitialize` do not change the `mesgPrintNumber`.
2. If `messageNumber` does not correspond to a valid message or group of messages, WIT displays a warning.

Exceptions to General Error Conditions

- Need not be preceeded by a call to `witInitialize`.

Error Conditions

- An undefined `messageNumber` is specified.

Example

```
witReturnCode rc;
rc = witSetMesgPrintNumber ( theWitRun,
     WitTRUE, WitINFORMATIONAL_MESSAGES, WitFALSE);
rc = witSetMesgPrintNumber ( theWitRun, WitTRUE, 326,
              WitTRUE )
```

All informational messages, except message 326, will be displayed without message numbers. Informational messages generated by `witSetMesgPrintNumber` will not be displayed.

witSetMesgStopRunning

```
witReturnCode witSetMesgStopRunning
(    WitRun  * const theWitRun,
     const witBoolean quiet,
     const int messageNumber,
     const witBoolean mesgStopRunning );
```

Description

Indicates if WIT should stop running or have control passed back to the application program after a severe or fatal message.

`theWitRun`

Identifies the WIT problem to be used by this function.

`quiet`

Indicates if the display of informational messages is to be suppressed for this function invocation.

`messageNumber`

The mesgStopRunning attribute for this message will be changed. `WitSEVERE_MESSAGES` can be specified to change all severe messages.

`WitFATAL_MESSAGES` can be specified to change all fatal messages.

`mesgStopRunning`

`WitTRUE` indicates that WIT causes program execution to terminate after issuing the severe or fatal message by executing a C `exit` statement. `WitFALSE` indicates that control is returned to the application program after issuing severe or fatal message.

Usage Notes

1. Calls to `witInitialize` do not change `mesgStopRunning`.
2. Calls with `messageNumber` equal to `WitINFORMATIONAL_MESSAGES` or `WitWARNING_MESSAGES` are ignored.
3. If `messageNumber` does not correspond to a valid message or group of messages, WIT displays a warning.
4. If `messageNumber` corresponds to an informational or warning message, the attribute is set, but it has no effect.
5. If the application program is to regain control after a severe or fatal message was issued, then this attribute must be set to `WitFALSE`. After WIT has issued a severe or fatal message, WIT's internal data structures are no longer in a valid state, and no further WIT functions should be called (even with a different WitRun).

Exceptions to General Error Conditions

- Need not be preceeded by a call to `witInitialize`.

Example

```
witReturnCode rc;
rc = witSetMesgStopRunning ( theWitRun,
             WitTRUE, WitSEVERE_MESSAGES, WitFALSE);
```

WIT will not stop running after a severe message, but will return control to the application program. Informational messages generated by `witSetMesgStopRunning` will not be displayed.

witSetMesgThrowErrorExc

```
witReturnCode witSetMesgThrowErrorExc
(    WitRun * const  theWitRun,
     const witBoolean quiet,
     const int        messageNumber,
     const witBoolean mesgThrowErrorExc );
```

Description

Indicates whether or not WIT should throw an exception after a severe or fatal message.

`theWitRun`

Identifies the WIT problem to be used by this function.

`quiet`

Indicates if the display of informational messages is to be suppressed for this function invocation.

`messageNumber`

The mesgThrowErrorExc attribute for this message will be changed. `WitSEVERE_MESSAGES` can be specified to change all severe messages.

`WitFATAL_MESSAGES` can be specified to change all fatal messages.

`mesgThrowErrorExc`

`WitTRUE` indicates that WIT will throw an exception after issuing the severe or fatal message.

Usage Notes

1. Calls to `witInitialize` do not change `mesgThrowErrorExc`.
2. Calls with `messageNumber` equal to `WitINFORMATIONAL_MESSAGES` or `WitWARNING_MESSAGES` are ignored.
3. If `messageNumber` does not correspond to a valid message or group of messages, WIT displays a warning.
4. If `messageNumber` corresponds to an informational or warning message, the attribute is set, but it has no effect.

Exceptions to General Error Conditions

- Need not be preceeded by a call to `witInitialize`.

Example

See “Sample 4: checkData.C” on page 429.

witSetMesgTimesPrint

```
witReturnCode witSetMesgTimesPrint
(    WitRun  * const theWitRun,
     const witBoolean quiet,
     const int messageNumber,
     const int mesgTimesPrint );
```

Description

Sets the number of times WIT will display a message.

`theWitRun`

Identifies the WIT problem to be used by this function.

`quiet`

Indicates if the display of informational messages is to be suppressed for this function invocation.

`messageNumber`

The `mesgTimesPrint` attribute for this message will be changed.

`WitINFORMATIONAL_MESSAGES`, `WitWARNING_MESSAGES`, or `WitSEVERE_MESSAGES` can be specified to change all informational, warning, or severe messages, respectively.

`mesgTimesPrint`

Indicates the number of times the message is to be printed. UCHAR_MAX can be specified to indicate that the message should always be printed. UCHAR_MAX is defined in the ANSI c file `limits.h.` Zero indicates that the message should never be printed.

Usage Notes

1. Calls to `witInitialize` does not change `mesgTimesPrint`.
2. If `messageNumber` does not correspond to a valid message or group of messages, WIT displays a warning.
3. Calling `witSetMesgTimesPrint` with `WitINFORMATIONAL_MESSAGES` as `messageNumber` and 0 as `mesgTimesPrint` will turn off printing of all informational messages. Note that the output of such functions as `witWriteExecSched` consists of informational messages, so this will cause these functions will create empty files. This applies to the following functions:
   - `witDisplayData`
   - `witWriteCriticalList`
   - `witWriteExecSched`
   - `witWriteReqSched`
   - `witWriteShipSched`

Error Conditions

- An undefined `messageNumber` is specified.
- mesgTimesPrint < 0
- mesgTimesPrint > UCHAR_MAX (See above). The value of UCHAR_MAX is platform dependent, but e.g., on AIX, its value is 255.

Exceptions to General Error Conditions

- Need not be preceeded by a call to `witInitialize`.

Example

```
witReturnCode rc;
rc = witSetMesgTimesPrint ( theWitRun,
             WitTRUE, WitINFORMATIONAL_MESSAGES, 0);
rc = witSetMesgTimesPrint ( theWitRun,
             WitTRUE, 167, UCHAR_MAX );
rc = witSetMesgTimesPrint (theWitRun, WitTRUE, 180, 10 );
```

WIT will not display any informational messages, except for message 167 and 180. Message 167 will always be displayed, and message 180 will only be displayed 10 times. Informational messages generated by `witSetMesgTimesPrint` will not be displayed.

Double Precision Functions

Consider the following API function:

```
witReturnCode witSetPartSupplyVol (
     WitRun * const      theWitRun,
     const char * const  partName,
     const float * const supplyVol);
```

This function (described earlier in this chapter) sets the value of the supplyVol attribute of the specified part to match the value given by the `supplyVol` argument. The type of the supplyVol attribute is considered to be “vector of floats”. However, as indicated in the note on page 97, WIT stores such attributes in double precision, and so the supplyVol attribute is stored internally as a vector of doubles. Note that the `supplyVol` argument to this function is a vector of floats, not doubles. Thus the `witSetPartSupplyVol` function does a type conversion, converting the values in the `supplyVol` argument from floats into doubles and storing them in the supplyVol attribute.

As an alternative, one can avoid this type conversion by using the following function:

```
witReturnCode witSetPartSupplyVolDbl (
     WitRun * const       theWitRun,
     const char * const   partName,
     const double * const supplyVol);
```

This function works in the same way as `witSetPartSupplyVol`, but the `supplyVol` argument is a vector of doubles and so WIT doesn’t need to do any type conversion when it sets the value of the supplyVol attribute to match the value of this argument.

In similar vein, consider the following API function:

```
witReturnCode witGetPartSupplyVol (
     WitRun * const     theWitRun,
     const char * const partName,
     float * *          supplyVol);
```

This function retrieves the value of the supplyVol attribute of a part. Specifically, it sets the vector given by `(* supplyVol)` to match the value of the supplyVol attribute. This involves type conversion in the opposite direction from the case above, converting a vector of doubles (the supplyVol attribute) into a vector of floats (the `(* supplyVol)` argument).

One can avoid this type conversion by using the following function:

```
witReturnCode witGetPartSupplyVolDbl (
     WitRun * const     theWitRun,
     const char * const partName,
     double * *         supplyVol );
```

This function works in the same way as `witGetPartSupplyVol`, but the `supplyVol` argument is the address of a vector of doubles and so WIT doesn’t need to do any type conversion when it sets the value of `(* supplyVol)` to match the value of the supplyVol attribute.

In general, for every API function that has at least one argument whose type involves “float”, there is a second API function whose corresponding argument(s) involve “double”. The functions with arguments involving “float” all do type conversion to or from internal values that are of types involving “double”. The functions with arguments involving “double” do not do type conversion. In each case, the “double” version of the function has the same name as the “float” version, but with the letters “Dbl” appended to the end of the name.

Examples of Double Precision Functions

Rather than a complete list of every double precision function, what follows is a collection of several illustrative examples.

```
witReturnCode witSetBopEntryExpAversionDbl (
     WitRun * const     theWitRun,
     const char * const producingOperationName,
     const int          bopEntryIndex,
     const double       expAversion);
```

Sets the expAversion of a BOP entry to the specified double precision value.

```
witReturnCode witGetBopEntryExpAversionDbl (
     WitRun * const     theWitRun,
     const char * const producingOperationName,
     const int          bopEntryIndex,
     double *           expAversion );
```

Stores the current value of the expAversion of a BOP entry in the specified double precision variable.

```
witReturnCode witFocusShortageVolDbl (
     WitRun * const theWitRun,
     int *          lenList,
     char * * *     partList
     double * * *   focusShortageVolList );
```

Retrieves the entire Focussed Shortage Schedule. The focussed shortageVols are retrieved as a list of vectors of type double.

```
witReturnCode witGetDemandExecVolPipDbl (
     WitRun * const     theWitRun,
     const char * const demandedPartName,
     const char * const demandName,
     const int          shipPeriod,
     int *              lenLists,
     char * * *         operationNameList,
     int * *            execPeriodList,
     double * *         peggedExecVolList);
```

Retrieves the post-implosion execVol pegging associated with a specific demand in a specific shipment period. The peggedExecVols are retrieved as a list of doubles.

```
witReturnCode witIncHeurAllocDbl (
   WitRun * const     theWitRun,
   const char * const demandedPartName,
   const char * const demandName,
   const int          shipPeriod,
   const double       desIncVol,
   double *           incVol);
```

Increments heuristic allocation. The desired increment to the shipVol is given as a double and the actual achieved increment to the shipVol is retrieved as a double.

Example Application Code

```
double   supVolInDbl[] = {1000000000017., 0};
float *  supVolOutFlt;
double * supVolOutDbl;

witSetNPeriods (theWitRun, 2);

witAddPart (theWitRun, "Mat1", WitMATERIAL);

witSetPartSupplyVolDbl (
   theWitRun, "Mat1",   supVolInDbl);

witGetPartSupplyVol (
   theWitRun, "Mat1", & supVolOutFlt);

witGetPartSupplyVolDbl (
   theWitRun, "Mat1", & supVolOutDbl);

printf (
   "supplyVol in float:  %13.0f\n"
   "supplyVol in double: %13.0f\n",
   supVolOutFlt[0],
   supVolOutDbl[0]);

free (supVolOutFlt);
free (supVolOutDbl);
```

The output of this code fragment might be the following;

```
supplyVol in float:   999999995904
supplyVol in double: 1000000000017
```

Usage Note

1. If the data of a WitRun has been specified using double precision and the application program calls `witWriteData`, it may be appropriate to set the global boolean attribute highPrecisionWD to TRUE. (See “highPrecisionWD” on page 104 and “witWriteData” on page 300.)

External Optimizing Implosion Functions

witStartExtOpt

```
witReturnCode witStartExtOpt (
   WitRun * const theWitRun);
```

Description

Initiates external optimizing implosion: Constructs an LP/MIP formulation of the implosion problem and puts external optimizing implosion into an active state. See “External Optimizing Implosion” on page 59.

`theWitRun`

Identifies the WIT problem to be used by this function.

Error Conditions

- External optimizing implosion must be inactive.
- The following global boolean attributes must be FALSE:
  - accAfterOptImp (on page 98)
  - computeCriticalList (on page 99)
  - compPrices (on page 99)

Code Example: Function extOptLp

The following function solves the implosion problem for its WitRun argument using external optimizing implosion, treating the optimization problem as an LP problem. The solvers used are COIN’s CLP and OSI. The function returns true, if an optimal solution was found; otherwise it returns false. The function invokes two other example functions specified elsewhere in this Guide:

- `passLpProb` (Page 399)
- `passSolution` (Page 405)

```
#include <wit.h>
#include <OsiClpSolverInterface.hpp>

bool extOptLp (WitRun * theWitRun)
   {
   OsiSolverInterface * theOsiSI;
   bool                 optFound;

   theOsiSI = new OsiClpSolverInterface;

   witStartExtOpt (theWitRun);

   passLpProb (theOsiSI, theWitRun);

   theOsiSI->initialSolve ();

   optFound = theOsiSI->isProvenOptimal ();

   if (optFound)
      {
      passSolution      (theWitRun, theOsiSI);

      witFinishExtOpt   (theWitRun);
      }
   else
      witShutDownExtOpt (theWitRun);

   delete theOsiSI;

   return optFound;
   }
```

witFinishExtOpt

```
witReturnCode witFinishExtOpt (
   WitRun * const theWitRun);
```

Description

Concludes external optimizing implosion: Constructs an implosion solution from the solution to the LP/MIP problem and puts external optimizing implosion into an inactive state. See “External Optimizing Implosion” on page 59.

`theWitRun`

Identifies the WIT problem to be used by this function.

Error Conditions

- External optimizing implosion must be active.
- Must be preceeded by a call to `witSetExtOptSoln` or `witSetExtOptSolnDbl`.

Code Example

See `expOptLp` on page 393.

witShutDownExtOpt

```
witReturnCode witShutDownExtOpt (
   WitRun * const theWitRun);
```

Description

Aborts external optimizing implosion: Puts external optimizing implosion into an inactive state without constructing an implosion solution. See “External Optimizing Implosion” on page 59.

`theWitRun`

Identifies the WIT problem to be used by this function.

Error Conditions

- External optimizing implosion must be active.

Code Example

See `expOptLp` on page 393.

witGetExtOptLpProb / witGetExtOptLpProbDbl

```
witReturnCode witGetExtOptLpProb (
   WitRun * const theWitRun,
   int *          numcols,
   int *          numrows,
   int * *        start,
   int * *        index,
   float * *      value,
   float * *      collb,
   float * *      colub,
   float * *      obj,
   float * *      rowlb,
   float * *      rowub);

witReturnCode witGetExtOptLpProbDbl (
   WitRun * const theWitRun,
   int *          numcols,
   int *          numrows,
   int * *        start,
   int * *        index,
   double * *     value,
   double * *     collb,
   double * *     colub,
   double * *     obj,
   double * *     rowlb,
   double * *     rowub);
```

Description

Each of these functions passes back to the application program a representation of the LP problem constructed by external optimizing implosion, or in MIP mode, the LP relaxation of the MIP problem. See “External Optimizing Implosion” on page 59. `witGetExtOptLpProb` has arguments of type `float * *`, while `witGetExtOptLpProbDbl` has arguments of type `double * *`.

`theWitRun`

Identifies the WIT problem to be used by this function.

`numcols`

On return `(* numcols)` is the number of columns in the LP problem.

`numrows`

On return `(* numrows)` is the number of rows in the LP problem.

`start`

On return,

`0 = (* start)[0] ≤ (* start)[1] ≤ ... ≤ (* start)[numcols]`

On return, for j = 0, ..., numcols -1:

for `k = (* start)[j], ..., (* start)[j+1] - 1`:

`j` is the column index of matrix element #`k`.

`index`

On return, for `k = 0, ..., (* start)[numcols]` - 1:

`(* index)[k]` is the row index of matrix element #`k`.

`value`

On return, for `k = 0, ..., (* start)[numcols]` - 1:

`(* value)[k]` is the value of matrix element #`k`.

`collb`

On return, for `j = 0, ..., numcols - 1:`

`(* collb)[j]` is the lower bound on column #`j`.

`colub`

On return, for `j = 0, ..., numcols - 1:`

`(* colub)[j]` is the upper bound on column #`j`.

`obj`

On return, for `j = 0, ..., numcols - 1:`

`(* obj)[j]` is the objective function coefficient on column #`j`.

`rowlb`

On return, for `i = 0, ..., numrows - 1:`

`(* rowlb)[i]` is the lower bound on row #`i`.

`rowub`

On return, for `i = 0, ..., numrows - 1:`

`(* rowub)[i]` is the upper bound on row #`i`.

Usage Notes

- It is the responsibility of the application program to free the returned vectors.
- The data retrieved by `witGetExtOptLpProbDbl` is represented in a form suitable to be passed to the COIN OSI function `OsiSolverInterface::loadProblem`.
- Note that WIT's LP formulation is a maximization problem and so this fact will need to be communicated to the solver.

Error Conditions

- External optimizing implosion must be active.

Code Example: Function passLpProb

The following function takes as arguments an OsiSolverInterface from COIN OSI and a WitRun from WIT and copies the LP problem from the WitRun into the OsiSolverInterface. External optimizing implosion must be active for the WitRun.

```
#include <wit.h>
#include <OsiSolverInterface.hpp>

void passLpProb (
      OsiSolverInterface * theOsiSI,
      WitRun *             theWitRun)
   {
   int      numcols;
   int      numrows;
   int *    start;
   int *    index;
   double * value;
   double * collb;
   double * colub;
   double * obj;
   double * rowlb;
   double * rowub;

   witGetExtOptLpProbDbl (
        theWitRun,
      & numcols,
      & numrows,
      & start,
      & index,
      & value,
      & collb,
      & colub,
      & obj,
      & rowlb,
      & rowub);
theOsiSI->
   loadProblem (
      numcols,
      numrows,
      start,
      index,
      value,
      collb,
      colub,
      obj,
      rowlb,
      rowub);

free (start);
free (index);
free (value);
free (collb);
free (colub);
free (obj);
free (rowlb);
free (rowub);

theOsiSI->setObjSense (-1.0);
   //
   // Tells OSI to maximize the objective function.
}
```

For examples of code that uses `passLpProb`, see `expOptLp` on page 393 and `expOptMip` on page 402.

witGetExtOptIntVarIndices

```
witReturnCode witGetExtOptIntVarIndices (
   WitRun * const theWitRun,
   int * *        indices,
   int *          len);
```

Description

In MIP mode, this function passes back to the application program the set of column indices of the integer variables in the MIP problem constructed by external optimizing implosion. See “External Optimizing Implosion” on page 59.

`theWitRun`

Identifies the WIT problem to be used by this function.

`indices`

On return,
`{(* indices)[0], ..., (* indices)[(* len)-1]}`
is the set of indices of all integer variables in the MIP problem.

`len`

On return, `(* len)` is the number of integer variables in the MIP problem.

Usage Notes

- It is the responsibility of the application program to free the returned vector.
- The data retrieved by this function is represented in a form suitable to be passed to the COIN OSI function `OsiSolverInterface::setInteger.`
- If optimizing implosion is not in MIP mode, `(* len)` = 0.

Error Conditions

- External optimizing implosion must be active.

Code Example #1: Function passIntVarIndices

The following function takes as arguments an OsiSolverInterface from COIN OSI and a WitRun from WIT, obtains from the WitRun the set of column indices of the integer variables in the MIP problem, and passes this set to the OsiSolverInterface. External optimizing implosion must be active for the WitRun.

```
#include <wit.h>
#include <OsiSolverInterface.hpp>

void passIntVarIndices (
      OsiSolverInterface * theOsiSI,
      WitRun *             theWitRun)
   {
   int * indices;
   int   len;

   witGetExtOptIntVarIndices (
        theWitRun,
      & indices,
      & len);

   theOsiSI->setInteger (indices, len);

   free (indices);
   }
```

Code Example #2: Function extOptMip

The following function solves the implosion problem for its WitRun argument using external optimizing implosion, treating the optimization problem as a MIP problem. The solvers used are COIN’s CLP, OSI, and CBC. The function returns true, if an optimal solution was found; otherwise it returns false. The function invokes three other example functions specified elsewhere in this Guide:

- `passLpProb` (Page 399)
- `passIntVarIndices` (Page 402)
- `passSolution` (Page 405)

```
#include <wit.h>
#include <OsiClpSolverInterface.hpp>
#include <CbcModel.hpp>

bool extOptMip (WitRun * theWitRun)
   {
   OsiSolverInterface * theOsiSI;
   CbcModel *           theCbcModel;
   bool                 optFound;

   theOsiSI    = new OsiClpSolverInterface;

   theCbcModel = new CbcModel (* theOsiSI);

   delete theOsiSI;

   theOsiSI    = NULL;

   witStartExtOpt (theWitRun);

   passLpProb        (theCbcModel->solver (), theWitRun);
   passIntVarIndices (theCbcModel->solver (), theWitRun);

   theCbcModel->branchAndBound ();

   optFound = theCbcModel->solver ()->isProvenOptimal ();

   if (optFound)
      {
      passSolution (theWitRun, theCbcModel->solver ());

      witFinishExtOpt (theWitRun);
      }
   else
      witShutDownExtOpt (theWitRun);

   delete theCbcModel;

   return optFound;
   }
```

witSetExtOptSoln / witSetExtOptSolnDbl

```
witReturnCode witSetExtOptSoln (
   WitRun * const      theWitRun,
   const float * const colsol);

witReturnCode witSetExtOptSolnDbl (
   WitRun * const       theWitRun,
   const double * const colsol);
```

Description

Each of these functions loads into WIT the optimal primal solution to the LP/MIP problem that was computed by external optimizing implosion. See “External Optimizing Implosion” on page 59. `witSetExtOptSoln` has an argument of type `const float * const`, while `witSetExtOptSolnDbl` has an argument of type `const double * const.`

`theWitRun`

Identifies the WIT problem to be used by this function.

`colsol`

Let `numcols` be the number of columns in the LP problem.

For `j = 0, ..., numcols-1:`

`colsol[j]` is the optimal primal solution value for column #`j`.

Usage Notes

- When using `witSetExtOptSolnDbl`, the `colsol` argument corresponds to the data retrieved by the COIN OSI function `OsiSolverInterface::getColSolution`.

Error Conditions

- External optimizing implosion must be active.
- It is the responsibility of the application program to ensure that `colsol` specifies a feasible solution to the LP/MIP problem. Normally, the solution should be optimal, but a non-optimal feasible solution is also acceptable. If the solution is not feasible, a severe error will be issued when `witFinishExtOpt` is invoked.

Code Example: Function passSolution

The following function takes as arguments a WitRun from WIT and an OsiSolverInterface from COIN OSI and copies the LP/MIP solution from the OsiSolverInterface into theWitRun. External optimizing implosion must be active for the WitRun.

```
#include <wit.h>
#include <OsiSolverInterface.hpp>

void passSolution (
      WitRun *             theWitRun,
      OsiSolverInterface * theOsiSI)
   {
   const double * colsol;

   colsol = theOsiSI->getColSolution ();

   witSetExtOptSolnDbl (theWitRun, colsol);
   }
```

For examples of code that uses `passSolution` see `expOptLp` on page 393 and `expOptMip` on page 402.

witGet*VarIndex

```
witReturnCode witGetBomEntryNonSubVarIndex (
   WitRun * const     theWitRun,
   const char * const consumingOperationName,
   const int          bomEntryIndex,
   const int          thePer,
   int *              nonSubVarIndex);

witReturnCode witGetDemandCumShipSlbvVarIndex (
   WitRun * const     theWitRun,
   const char * const partName,
   const char * const demandName,
   const int          thePer,
   int *              cumShipSlbvVarIndex);

witReturnCode witGetDemandCumShipVarIndex (
   WitRun * const     theWitRun,
   const char * const partName,
   const char * const demandName,
   const int          thePer,
   int *              cumShipVarIndex);

witReturnCode witGetDemandShipVarIndex (
   WitRun * const     theWitRun,
   const char * const partName,
   const char * const demandName,
   const int          thePer,
   int *              shipVarIndex);

witReturnCode witGetOperationExecSlbvVarIndex (
   WitRun * const     theWitRun,
   const char * const operationName,
   const int          thePer,
   int *              execSlbvVarIndex);

witReturnCode witGetOperationExecVarIndex (
   WitRun * const     theWitRun,
   const char * const operationName,
   const int          thePer,
   int *              execVarIndex);

witReturnCode witGetPartScrapVarIndex (
   WitRun * const     theWitRun,
    const char * const partName,
    const int          thePer,
    int *              scrapVarIndex);

 witReturnCode witGetPartStockSlbvVarIndex (
    WitRun * const     theWitRun,
    const char * const partName,
    const int          thePer,
    int *              stockSlbvVarIndex);

 witReturnCode witGetPartStockVarIndex (
    WitRun * const     theWitRun,
    const char * const partName,
    const int          thePer,
    int *              stockVarIndex);

 witReturnCode witGetSubsBomEntrySubVarIndex (
    WitRun * const     theWitRun,
    const char * const consumingOperationName,
    const int          bomEntryIndex,
    const int          subsBomEntryIndex,
    const int          thePer,
    int *              subVarIndex);
```

Description

These are the column index functions for external optimizing implosion. (See “External Optimizing Implosion” on page 59.) Each of these functions passes back to the application program the column index of a particular variable in the LP/MIP problem. For example, `witGetPartStockVarIndex` passes back to the application program the column index of the stock variable for a particular part in a particular period.

Arguments for `witGetPartStockVarIndex`:

`theWitRun`

Identifies the WIT problem to be used by this function.

`partName`

The partName of the part with which the stock variable is associated.

`thePer`

The period with which the stock variable is associated.

`stockVarIndex`

On return `(* stockVarIndex)` is the column index of the stock variable associated with the part in the period.

The following table lists each column index function along with the type of LP/MIP variable whose column index it retrieves and the type of WIT data object with which the variable is associated:

TABLE 7 **Column Index Functions**

Function Name	LP/MIP Variable Type	Object Type
`witGetPartScrapVarIndex`	Scrap	Part
`witGetPartStockVarIndex`	Stock	Part
`witGetDemandShipVarIndex`	Shipment	Demand
`witGetDemandCumShipVarIndex`	Cumulative Shipment	Demand
`witGetOperationExecVarIndex`	Execution	Operation
`witGetBomEntryNonSubVarIndex`	Non-substitution	BOM Entry
`witGetSubsBomEntrySubVarIndex`	Substitution	Substitute
`witGetPartStockSlbvVarIndex`	Stock SLBV	Part
`witGetDemandCumShipSlbvVarIndex`	Cumulative Shipment SLBV	Demand
`witGetOperationExecSlbvVarIndex`	Execution SLBV	Operation

Usage Notes

- If there is no LP/MIP variable corresponding to the arguments, -1 is passed back as the column index.

Error Conditions

- External optimizing implosion must be active.
- The arguments must specify an existing WIT data object. For example, in `witGetPartStockVarIndex`, the `partName` argument must specify an existing part.
- `thePer` must be in the range: `0 ≤ thePer < nPeriods`.
- In `witGetPartStockVarIndex` and `witGetPartStockSlbvVarIndex`, the part specified by the `partName` argument must have partCategory = material.

Code Example: Function setScrapUB

The following function takes as arguments an OsiSolverInterface from COIN OSI, a WitRun from WIT, the name of a part, a period, and a double precision value `scrapUB`. It is assumed that external optimizing implosion is active for the WitRun and that the LP/MIP problem has already been passed from the WitRun to the OsiSolverInterface. The function imposes an upper bound of value `scrapUB` on the scrap variable for the part in the period.

```
#include <wit.h>
#include <OsiSolverInterface.hpp>

void setScrapUB (
      OsiSolverInterface * theOsiSI,
      WitRun *             theWitRun,
      const char *         partName,
      int                  thePer,
      double               scrapUB)
   {
   int scrapVarIndex;

   witGetPartScrapVarIndex (
        theWitRun,
        partName,
        thePer,
      & scrapVarIndex);

   theOsiSI->setColUpper (scrapVarIndex, scrapUB);
   }
```

witGet*ConIndex

```
witReturnCode witGetBomEntrySubConIndex (
   WitRun * const     theWitRun,
   const char * const consumingOperationName,
   const int          bomEntryIndex,
   const int          thePer,
   int *              subConIndex);

witReturnCode witGetDemandCumShipSlbConIndex (
   WitRun * const     theWitRun,
   const char * const partName,
   const char * const demandName,
   const int          thePer,
   int *              cumShipSlbConIndex);

witReturnCode witGetDemandShipConIndex (
   WitRun * const     theWitRun,
   const char * const partName,
   const char * const demandName,
   const int          thePer,
   int *              shipConIndex);

witReturnCode witGetOperationExecSlbConIndex (
   WitRun * const     theWitRun,
   const char * const operationName,
   const int          thePer,
   int *              execSlbConIndex);

witReturnCode witGetPartResourceConIndex (
   WitRun * const     theWitRun,
   const char * const partName,
   const int          thePer,
   int *              resourceConIndex);

witReturnCode witGetPartStockSlbConIndex (
   WitRun * const     theWitRun,
   const char * const partName,
   const int          thePer,
   int *              stockSlbConIndex);
```

Description

These are the row index functions for external optimizing implosion. (See "External Optimizing Implosion" on page 59.) Each of these functions passes back to the application program the row index of a particular constraint in the LP/MIP problem. For example, `witGetPartResourceConIndex` passes back to the application program the column index of the resource allocation for a particular part in a particular period.

Arguments for `witGetPartResourceConIndex`:

`theWitRun`

Identifies the WIT problem to be used by this function.

`partName`

The partName of the part with which the resource allocation constraint is associated.

`thePer`

The period with which the resource allocation constraint is associated.

`resourceConIndex`

On return `(* resourceConIndex)` is the row index of the resource allocation constraint associated with the part in the period.

The following table lists each row index function along with the type of LP/MIP constraint whose row index it retrieves and the type of WIT data object with which the constraint is associated:

TABLE 8 **Row Index Functions**

Function Name	LP/MIP Constraint Type	Object Type
`witGetPartResourceConIndex`	Resource Allocation	Part
`witGetDemandShipConIndex`	Shipment	Demand
`witGetBomEntrySubConIndex`	Substitution	BOM Entry
`witGetPartStockSlbConIndex`	Stock SLB	Part
`witGetDemandCumShipSlbConIndex`	Cumulative Shipment SLB	Demand
`witGetOperationExecSlbConIndex`	Execution SLB	Operation

Usage Notes

- If there is no LP/MIP constraint corresponding to the arguments, -1 is passed back as the row index.

Error Conditions

- External optimizing implosion must be active.
- The arguments must specify an existing WIT data object. For example, in `witGetPartResourceConIndex`, the `partName` argument must specify an existing part.
- `thePer` must be in the range: `0 ≤ thePer < nPeriods`.
- In `witGetPartStockSlbConIndex`, the part specified by the `partName` argument must have partCategory = material.

Code Example

These functions are similar to the `witGet*VarIndex` functions. See `setScrapUB` on page 409.

APPENDIX A API Sample Code

Sample 1

```
/****************************************************************************
 *
 * Sample WIT API Program
 *

 * 5799-QYH
 * (C) Copyright IBM Corp. 1996 All Rights Reserved
 *
 * This basic program demonstrates the use of a number of API
 * calls.  It loads data into the WIT model via API calls, implodes,
 * and writes the output.
 *
 ****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <wit.h>

void main (int argc, char * argv[])
{
  /* Setup the WitRun */
  WitRun * theWitRun;
  witNewRun( &theWitRun );
  witInitialize ( theWitRun );

  /* Set global attributes */
  witSetNPeriods    ( theWitRun, 4 );
  witSetExecEmptyBom( theWitRun, WitTRUE );
  witSetTitle       ( theWitRun, “quote mark: \” back slash: \\” );

  /* Create objects */

  /* Create part A, operation A, and connecting BOP */
  {
  float stockCost[] = { 50., 50., 50., 50. };
  float scrapCost[] = { 50., 50., 50., 50. };
  witAddPartWithOperation( theWitRun, “A” );
  witSetPartStockCost( theWitRun, “A”, stockCost );
  witSetPartScrapCost( theWitRun, “A”, scrapCost );
}

/* Create part B */
{
float stockCost[] = {  1.,   1.,  1.,  1. };
float scrapCost[] = { 10.,  10., 10., 10. };
float supplyVol    [] = {  0., 100.,  0., 50. };
witAddPart             ( theWitRun, “B”, WitMATERIAL );
witSetPartSupplyVol    ( theWitRun, “B”, supplyVol     );
witSetPartStockCost( theWitRun, “B”, stockCost );
witSetPartScrapCost( theWitRun, “B”, scrapCost );
}

/* Create capacity C */
{
float supplyVol    [] = {  30.,  30.,  30.,  30. };
witAddPart             ( theWitRun, “C”, WitCAPACITY );
witSetPartSupplyVol    ( theWitRun, “C”, supplyVol     );
}

/* Create part E */
{
float stockCost[] = {   1.,   1.,   1.,   1. };
float scrapCost[] = {  10.,  10.,  10.,  10. };
float supplyVol    [] = {  25.,  25.,  25.,  25. };
witAddPart             ( theWitRun, “E”, WitMATERIAL );
witSetPartSupplyVol    ( theWitRun, “E”, supplyVol     );
witSetPartStockCost( theWitRun, “E”, stockCost );
witSetPartScrapCost( theWitRun, “E”, scrapCost );
}

/* Create demand F on part A */
witAddDemand( theWitRun, “A”, “F” );

/* Create operation D */
witAddOperation( theWitRun, “D” );

/* Create Bill-of-manufacturing entries */
witAddBomEntry( theWitRun, “A”, “C” );
witAddBomEntry( theWitRun, “A”, “B” );
witAddBomEntry( theWitRun, “D”, “E” );

/* Create Substitute BOM Entry where part E may be used in place */
/* of part B in the BOM entry representing the consumption of    */
/* part B by operation A.                                         */
witAddSubsBomEntry( theWitRun, “A”, 1, “E” );

/* Create Bill-of-process entries */
witAddBopEntry( theWitRun, “D”, “A” );

/* Set object attributes */
/* Set part A attributes */
{
float softLowerBound[] = { 10., 10., 10., 10. };
float hardUpperBound[] = { 30., 30., 20., 20. };
float supplyVol     [] = { 17.,  0.,  0.,  0. };
witSetPartStockBounds( theWitRun, “A”,
                       NULL, softLowerBound, hardUpperBound );
witSetPartSupplyVol  ( theWitRun, “A”, supplyVol );
}

/* Set demand F on part A attributes */
{
float demandVol[] = {   50.,   60.,   70.,   80. };
float shipReward[] = { 1000., 1000., 1000., 1000. };
float cumShipReward[] = {   10.,   10.,   10.,   10. };
witSetDemandDemandVol        ( theWitRun, “A”, “F”, demandVol );
witSetDemandShipReward   ( theWitRun, “A”, “F”, shipReward );
witSetDemandCumShipReward( theWitRun, “A”, “F”, cumShipReward );
}

/* Set operation A attributes */
{
int yield[] = {   95,   95,   95,   95 };
witSetOperationYield( theWitRun, “A”, yield );
}

/* Set BOM Entry attributes */
{
float offset[] = { 1., 1., 1., 1. };
witSetBomEntryOffset( theWitRun, “A”, 1, offset );
}

/* Set substitute BOM Entry attributes */
witSetSubsBomEntryLatestPeriod( theWitRun, “A”, 1, 0, 2 );

/* Set BOP Entry attributes */
{
float productRate[] = { 2., 2., 2., 2. };
witSetBomEntryProductRate( theWitRun, “A”, 0, productRate);
}



/* Perform Implosion and write production and shipment schedule */
witOptImplode( theWitRun );
witWriteExecSched( theWitRun, WitSTDOUT, WitBSV );
witWriteShipSched( theWitRun, WitSTDOUT, WitBSV );

/* Get and print a few attribute values for part B */
{
  int     i, nPeriods;
  float * supplyVol;
  float * consVol;
  float * stockVol;
  float * excessVol;
  witGetNPeriods     ( theWitRun,      &nPeriods  );
  witGetPartSupplyVol( theWitRun, “B”, &supplyVol );
  witGetPartConsVol  ( theWitRun, “B”, &consVol   );
  witGetPartStockVol ( theWitRun, “B”, &stockVol  );
  witGetPartExcessVol( theWitRun, “B”, &excessVol );
  for( i=0; i<nPeriods; i++ )
     printf( “part B: supplyVol[%d]=%f, consVol[%d]  =%f\n”
             “        stockVol[%d] =%f, excessVol[%d]=%f\n”,
             i, supplyVol[i],
             i, consVol[i],
             i, stockVol[i],
             i, excessVol[i] );
  free( supplyVol );
  free( consVol   );
  free( stockVol  );
  free( excessVol );
  }

  witDeleteRun( theWitRun );

} /* main */
}
```

Sample 2

```
/******************************************************************************
 *
 * Sample WIT API Program
 *

 * 5799-QYH
 * (C) Copyright IBM Corp. 1996 All Rights Reserved
 *
 * This program determines the number of additional demand streams which can
 * be satisfied when the supply of critical parts is increased by a
 * given percentage.
 *
 * The steps are:
 *    - Read a WIT data file
 *    - Set FSS parameters
 *    - Run the heuristic implosion
 *    - Count number of demand streams which were not met
 *    - Get focussed shortage schedule
 *    - Increase supply of short parts
 *    - Rerun the heuristic implosion
 *    - Count number of demand streams which were not met
 *    - Compare number of demands which were not met before and after
 *      the focused shortage part supply was increased
 *
 ******************************************************************************/

#include <stdlib.h>
#include <wit.h>

static
int numUnmetDemands(
    WitRun * const theWitRun,              /* The WIT Environment            */
    const int nPeriods,                    /* Number of periods              */
    const int nParts,                      /* Number of parts                */
    char **  partList );                   /* List of part names             */

/******************************************************************************
 * Main Program
 * The name of the wit data file must be specified on command line.
 ******************************************************************************/

void main (int argc, char * argv[]) {

   int nPeriods;                          /* Number of periods in model     */
   int nParts;                            /* Number of parts in model       */
   int shortPartListLen;                  /* Number of parts with shortages */
   int unmetDemand1, unmetDemand2;        /* Count of unsatisfied demands   */
   float ** focusShortVolList;            /* Magnitude of part shortages.   */
   char ** partList;                      /* List of all parts.             */
char ** shortPartList;               /* List of parts with shortages.  */
int i;                               /* Loop index                     */
WitRun *  theWitRun;                 /* Current Wit Run                */

/*
 * Make sure a wit.data file was specified on command line.
 */
if ( argc < 2 ) {
   printf( “usage: %s wit_data_file_name\n”,argv[0]);
   exit(1);
}

/*
 * Establish environment for WIT to run.
 */
witNewRun(&theWitRun);

/*
 * Send WIT messages to file wit.out, and write over an existing file.
 */
witSetMesgFileAccessMode( theWitRun, WitTRUE, “w” );
witSetMesgFileName( theWitRun, WitTRUE, “wit.out” );

/*
 * Initialize WIT
 */
witInitialize( theWitRun );

/*
 * Read WIT data file specified on command line.
 */
witReadData( theWitRun, argv[1] );

/*
 * Set FSS to focus mode; use universal focus.
 */
witSetUseFocusHorizons( theWitRun, WitTRUE );

/*
 * Get number of periods and list of part names.
 */
witGetNPeriods( theWitRun, &nPeriods );
witGetParts( theWitRun, &nParts, &partList );

/*
 * Set FSS horizon on all parts with demands.
 */
for(i=0;i<nParts;++i)
{
  int j;
  int demListLen;
```

```
    char ** demList;
    witGetPartDemands(theWitRun,partList[i],&demListLen,&demList);
    if(demListLen)
    {
      for(j=0;j<demListLen;j++)
      {
        witSetDemandFocusHorizon(
          theWitRun,
          partList[i],
          demList[j],
          nPeriods-1);
      }
      free(demList[j]);
    }
    free(demList);
  }

  /*
   * Invoke heuristic implosion.
   */
  witHeurImplode( theWitRun );

  /*
   * Count number of unmet demand streams and get objective
   * function values prior to adjusting supply.
   */
  unmetDemand1 = numUnmetDemands( theWitRun, nPeriods, nParts, partList);

  /* Get focussed shortage schedule. */
  witGetFocusShortageVol(theWitRun,
    &shortPartListLen,
    &shortPartList,
    &focusShortVolList);

  /* Increase supply by shortage amounts. */
  for(i=0;i<shortPartListLen;i++)
  {
    int t;
    float * supply;
    witGetPartSupplyVol(theWitRun,shortPartList[i],&supply);
    for(t=0;t<nPeriods;t++) supply[t]+= focusShortVolList[i][t];
    witSetPartSupplyVol(theWitRun,shortPartList[i],supply);
    free(supply);
    free(shortPartList[i]);
    free(focusShortVolList[i]);
  }
  free(shortPartList);
  free(focusShortVolList);

  /*
   * Re-implode with increased supply.
```

```
    */
   witHeurImplode( theWitRun );

   /*
    * Count number of unmet demand streams after supply changes.
    */
   unmetDemand2 = numUnmetDemands( theWitRun, nPeriods, nParts, partList );

   /*
    * Write result.
    */
   printf( “Before increasing supply:\n” );
   printf( “  Number of unmet demand streams : %d\n”, unmetDemand1  );
   printf( “After increasing supply:\n” );
   printf( “  Number of unmet demand streams : %d\n”, unmetDemand2  );

   /* Free dynamically allocated memory. */
   for(i=0;i<nParts;++i) free(partList[i]);
   free(partList);

   /* Free storage associated with the WIT environment */
   witDeleteRun( theWitRun );

   exit (0);

}  /* main */

/*******************************************************************************
 *
 * Count number of demand streams which are not satisfied.
 *
 *******************************************************************************/

int numUnmetDemands(
    WitRun * const theWitRun,           /* WIT environment                */
    const int nPeriods,                 /* Number of periods              */
    const int nParts,                   /* Number of parts                */
    char ** partList )                  /* List of part names             */
{

   int i;                                      /* Loop index                */
   int unmetDemands=0;                         /* Count of unmet demands    */

   /*
    * Loop once for each part.
    */
   for ( i=0; i<nParts; i++ ) {

      int nDemands;                           /* Number of demands on part */
      char ** demandList;                     /* List of demands on part   */
      int j,t;                                /* Loop indices              */
      /*
       * Get list of demands defined for part.
       */
      witGetPartDemands( theWitRun, partList[i], &nDemands, &demandList );

      /*
       * Loop once for each demand
       */
      for ( j=0; j<nDemands; j++ ) {

         float * shipq;
         float * demandq;
         witBoolean met;

         /*
          * Get demand and shipment quantity for part
          */
         witGetDemandDemandVol( theWitRun,
            partList[i], demandList[j], &demandq );
         witGetDemandShipVol( theWitRun, partList[i], demandList[j], &shipq );

         /*
          * Check to see if demand for any demand stream could not
          * be met. Increment count if demand can not be met.
          */
         met = WitTRUE;
         for ( t=0; t<nPeriods; t++ )
            if ( demandq[t] > shipq[t] ) {
               met = WitFALSE;
               break;
            }
         if ( !met ) unmetDemands++;

         /*
          * Free demand and shipment quantity storage.
          */
         free( demandq );
         free( shipq );

      } /* for ( j=0; j<nDemands; j++ ) */

      /*
       * Free demandList storage
       */
      for ( j=0; j<nDemands; j++ ) free( demandList[j] );
      free( demandList );

   }  /* for ( i=0; i<nParts; i++ ) */

   return unmetDemands;
```

```
}
```

Sample 3

```
/*********************************************************************
 *
 * Sample WIT API Program
 *

 * 5799-QYH
 * (C) Copyright IBM Corp. 1996 All Rights Reserved
 *
 * This program is an example of using operation nodes.
 *
 * It models a yield where the production of PartA has a yield
 * of 50% completed the first period, 30% completed in 2nd period, and
 * the remaining 20% are completed in the 3rd period.
 *
 * This is done with three BOP entries for PartA’s OperationA.  The
 * first entry uses the default offset of 0, and sets the productRate to
 * 0.5.  This models 50% of the production completing in the period
 * that the operation begins.
 *
 * Similarly, the a second BOP entry is added to PartA’s OperationA.
 * However, this entry sets its offset to -1 with a productRate of 0.3.
 * This reflects that 30% of the production will complete 1 period
 * after the operation begins.  Finally, a third BOP entry is set
 * with offset -2 and productRate 0.2 to reflect that the remaining 20%
 * of the production will complete 2 periods after the operation begins.
 *
 *********************************************************************/

#include <stdlib.h>
#include <wit.h>

/*
 * Function prototypes.
 */
void writeDemandAttributeValue(
   WitRun * const theWitRun,
   const char * const demandedPartName,
   const char * const demandName,
   witReturnCode (*witGetDemandAttribFunc)(
      WitRun * const theWitRun,
      const char * const demandedPartName,
      const char * const demandName,
      float ** attributeValue ),
   char * title);
void writePartAttributeValue(
   WitRun * const theWitRun,
   const char * const partName,
   witReturnCode (*witGetPartAttribFunc)(
      WitRun * const theWitRun,
      const char * const partName,
      float ** attributeValue ),
   char * title);
void writeOperationAttributeValue(
   WitRun * const theWitRun,
   const char * const operationName,
   witReturnCode (*witGetOperationAttribFunc)(
      WitRun * const theWitRun,
      const char * const operationName,
      float ** attributeValue ),
   char * title
);

/*****************************************************************
 * Main Program
 *****************************************************************/
void main (int argc, char * argv[])
{
   WitRun * theWitRun;
   int nPeriods = 5;       /* Number of periods in model   */

   /*
    * Establish environment for WIT to run.
    */
   witNewRun(&theWitRun);

   /*
    * Initialize WIT
    */
   witInitialize( theWitRun );

   /*
    * Set up wit global attributes.
    */
   witSetNPeriods( theWitRun, nPeriods );

   /*
    * Add the parts and operations
    */
   witAddPart( theWitRun, “PartA”, WitMATERIAL );
   {
     float stockCost[] = {   .01,   .01, .01, .01, .01 };
     witSetPartStockCost( theWitRun, “PartA”, stockCost );
   }

   witAddOperation( theWitRun, “OperationA” );

   witAddPart( theWitRun, “Component”, WitMATERIAL );
   {
     float supplyVol[] = { 100., 100., 0., 0., 0. };
```

```
  witSetPartSupplyVol( theWitRun, “Component”, supplyVol );
}

/*
 * Add demands
 */
witAddDemand( theWitRun, “PartA”, “Demand1” );
{
  float demandVol[] = { 100., 0., 0., 0., 0. };
  float cumShipReward[] = { 1., 1., 1., 1., 1. };
  witSetDemandDemandVol( theWitRun, “PartA”, “Demand1”, demandVol );
}

/*
 * Add BOMs (bill-of-manufacturing)
 */
witAddBomEntry( theWitRun, “OperationA”, “Component” );

/*
 * Add BOPs (bill-of-products)
 */
witAddBopEntry( theWitRun, “OperationA”, “PartA” );
{
  float productRate[] = {.5,.5,.5,.5,.5 };
  witSetBopEntryproductRate( theWitRun, “OperationA”, 0, productRate );
}
witAddBopEntry( theWitRun, “OperationA”, “PartA” );
{
  float offset[] = {-1.,-1.,-1.,-1.,-1. };
  float productRate[] = {.3,.3,.3,.3,.3 };
  witSetBopEntryOffset  ( theWitRun, “OperationA”, 1, offset );
  witSetBopEntryproductRate( theWitRun, “OperationA”, 1, productRate );
}
witAddBopEntry( theWitRun, “OperationA”, “PartA” );
{
  float offset[] = {-2.,-2.,-2.,-2.,-2. };
  float productRate[] = {.2,.2,.2,.2,.2 };
  witSetBopEntryOffset  ( theWitRun, “OperationA”, 2, offset );
  witSetBopEntryproductRate( theWitRun, “OperationA”, 2, productRate );
}

/*
 * Implode
 */
witOptImplode( theWitRun );

/*
 * Explode
 */
witMrp( theWitRun );
```

```
   /*
    * Turn WIT Messages Off
    */
   witSetMesgTimesPrint( theWitRun,WitTRUE,WitINFORMATIONAL_MESSAGES,0);

   /*
    * Write Results
    */
   writeDemandAttributeValue( theWitRun,
                              “PartA”,
                              “Demand1”,
                              witGetDemandDemandVol,
                              “DemandVol” );
   writeDemandAttributeValue( theWitRun,
                              “PartA”,
                              “Demand1”,
                              witGetDemandShipVol,
                              “ShipVol” );
   writeOperationAttributeValue( theWitRun,
                                 “OperationA”,
                                 witGetOperationExecVol,
                                 “ExecVol” );
   writePartAttributeValue( theWitRun,
                            “PartA”,
                            witGetPartProdVol,
                            “ProdVol” );
   writePartAttributeValue( theWitRun,
                            “PartA”,
                            witGetPartConsVol,
                            “ConsVol” );
   writePartAttributeValue( theWitRun,
                            “PartA”,
                            witGetPartStockVol,
                            “StockVol” );
   writePartAttributeValue( theWitRun,
                            “Component”,
                            witGetPartSupplyVol,
                            “SupplyVol” );
   writePartAttributeValue( theWitRun,
                            “Component”,
                            witGetPartConsVol,
                            “ConsVol” );
   writePartAttributeValue( theWitRun,
                           “Component”,
                           witGetPartStockVol,
                           “StockVol” );
   writePartAttributeValue( theWitRun,
                           “Component”,
                           witGetPartReqVol,
                           “ReqVol” );
}
```

```
/***************************************************************
 * Write demand attribute
 ***************************************************************/
void writeDemandAttributeValue(
   WitRun * const theWitRun,
   const char * const demandedPartName,
   const char * const demandName,
   witReturnCode (*witGetDemandAttribFunc)(
      WitRun * const theWitRun,
      const char * const demandedPartName,
      const char * const demandName,
      float ** attributeValue ),
   char * title
)
{
  int nPeriods,i;
  float * attributeValue;

  witGetNPeriods( theWitRun, &nPeriods );
  witGetDemandAttribFunc(
    theWitRun,
    demandedPartName,
    demandName,
    &attributeValue );

  printf("%-10s %-10s %-10s: ",demandedPartName, demandName, title);
  for ( i=0; i<nPeriods; i++ ) printf("%8.1f ",attributeValue[i] );
  printf("\n");

  free( attributeValue );
}

/***************************************************************
 * Write Part attribute
 ***************************************************************/
void writePartAttributeValue(
   WitRun * const theWitRun,
   const char * const partName,
   witReturnCode (*witGetPartAttribFunc)(
      WitRun * const theWitRun,
      const char * const partName,
      float ** attributeValue ),
   char * title
)
{
  int nPeriods,i;
  float * attributeValue;

  witGetNPeriods( theWitRun, &nPeriods );
  witGetPartAttribFunc( theWitRun, partName, &attributeValue );
  printf(“%-15s  %-15s: “,partName,title);
  for ( i=0; i<nPeriods; i++ ) printf(“%8.1f “,attributeValue[i] );
  printf(“\n”);

  free( attributeValue );
}

/*****************************************************************
 * Write Operation attribute
 *****************************************************************/
void writeOperationAttributeValue(
   WitRun * const theWitRun,
   const char * const operationName,
   witReturnCode (*witGetOperationAttribFunc)(
      WitRun * const theWitRun,
      const char * const operationName,
      float ** attributeValue ),
   char * title
)
{
  int nPeriods,i;
  float * attributeValue;

  witGetNPeriods( theWitRun, &nPeriods );
  witGetOperationAttribFunc( theWitRun, operationName, &attributeValue );

  printf(“%-15s  %-15s: “,operationName,title);
  for ( i=0; i<nPeriods; i++ ) printf(“%8.1f “,attributeValue[i] );
  printf(“\n”);

  free( attributeValue );
}
```

Sample 4: checkData.C

```
//------------------------------------------------------------------------------
// C++ Program “checkData.C”
//
// This program is intended as a demonstration of how to catch an error
// exception thrown by WIT.
// The program takes a file name as its command line argument and displays
// text indicating whether or not the file is a valid WIT data file.
//------------------------------------------------------------------------------

#include <wit.h>

#include <string>
#include <iostream>

//------------------------------------------------------------------------------

void invokeWit (const std::string & fileName);

void handleException (
   const std::string & fileName,
   WitErrorExc &       theWitErrorExc);

//------------------------------------------------------------------------------
```

```
int main (int argc, char * argv[])
   {
   std::string fileName;

   if (argc != 2)
      {
      std::cerr
         << “ERROR: Exactly one argument must be passed to checkData.”
         << std::endl
         << std::endl;

      exit (1);
      }

   fileName = argv[1];

   try
      {
      invokeWit (fileName);
      }

   catch (WitErrorExc & theWitErrorExc)
      {
      handleException (fileName, theWitErrorExc);
      }

   std::cout
      << std::endl
      << “File \””
      << fileName
      << “\” is a valid WIT data file.”
      << std::endl
      << std::endl;

   exit (0);
   }
```

```
void invokeWit (const std::string & fileName)
   {
   WitRun * theWitRun;

   witNewRun (& theWitRun);

   remove (“wit.msg”);

   witSetMesgFileName (
      theWitRun,
      WitTRUE,
      “wit.msg”);

   witSetMesgThrowErrorExc (
      theWitRun,
      WitFALSE,
      WitSEVERE_MESSAGES,
      WitTRUE);

   witSetMesgThrowErrorExc (
      theWitRun,
      WitFALSE,
      WitFATAL_MESSAGES,
      WitTRUE);

   witInitialize (theWitRun);

   witReadData (theWitRun, fileName.c_str ());

   witDeleteRun (theWitRun);
   }
```

```
void handleException (
      const std::string & fileName,
      WitErrorExc &       theWitErrorExc)
   {
   std::string throwerName;

   throwerName = theWitErrorExc.funcName ();

   if (throwerName == “witReadData”)
      if (theWitErrorExc.retCode () == WitSEVERE_RC)
         {
         std::cout
            << std::endl
            << “File \””
            << fileName
            << “\” is a NOT valid WIT data file.”
            << std::endl
            << std::endl;

         exit (0);
         }

   std::cerr
      << std::endl
      << “checkData is terminating due to a programming error.”
      << std::endl
      << std::endl;

   exit (1);
   }
```

Sample 5: newsVendor.C

The following program uses stochastic implosion to solve the simple news vendor problem given in Chapter 1, page 30.

```
//------------------------------------------------------------------------------
// Program: "newsVendor.C".
//
// A C++ WIT application program to illustrate stochastic implosion.
// This program uses stochastic implosion to solve a very simple instance of
// the news vendor problem.
//------------------------------------------------------------------------------

#include <wit.h>
#include <iostream>

//------------------------------------------------------------------------------
// Function declarations.
//------------------------------------------------------------------------------

void buildCoreProblem (WitRun * & theWitRun);
   //
   // Builds the core problem in theWitRun.

void enterStochData (WitRun * theWitRun);
   //
   // Enters the stochastic data into theWitRun.

void enterScenarioData (
      WitRun * theWitRun,
      int      theScen,
      float    probability,
      float    demandVolPer1);
   //
   // Enters the data for one scenario into theWitRun.
   // theScen       is the index       of the scenario.
   // probability   is the probability of the scenario.
   // demandVolPer1 is the period 1 demandVol for demand "Sell" in the
   //               scenario.

void displaySoln (WitRun * theWitRun);
   //
   // Displays the solution to the news vendor problem from theWitRun.
```

```
//------------------------------------------------------------------------------
// Main Program
//------------------------------------------------------------------------------

int main ()
   {
   WitRun * theWitRun;

   buildCoreProblem (theWitRun);
   enterStochData   (theWitRun);
   witStochImplode  (theWitRun);
   displaySoln      (theWitRun);
   witDeleteRun     (theWitRun);

   exit (0);
   }

//------------------------------------------------------------------------------

void buildCoreProblem (WitRun * & theWitRun)
   {
   float supplyVol [] = {500.0, 0.0};
   float execCost  [] = {  0.6, 0.6};
   float shipReward[] = {  1.0, 1.0};

   witNewRun              (& theWitRun);

   witSetMesgFileAccessMode (theWitRun, WitTRUE, "w");
   witSetMesgFileName       (theWitRun, WitTRUE, "newsVendor.log");

   witInitialize            (theWitRun);
   witSetNPeriods           (theWitRun, 2);

   witAddPart               (theWitRun, "Source",           WitCAPACITY);
   witAddOperation          (theWitRun, "Buy");
   witAddBomEntry           (theWitRun, "Buy",    "Source");
   witAddPart               (theWitRun, "Hold",             WitMATERIAL);
   witAddBopEntry           (theWitRun, "Buy",    "Hold");
   witAddDemand             (theWitRun, "Hold",   "Sell");

   witSetPartSupplyVol      (theWitRun, "Source",           supplyVol);
   witSetOperationExecCost  (theWitRun, "Buy",              execCost);
   witSetDemandShipReward   (theWitRun, "Hold",   "Sell",   shipReward);
   }
```

```
//------------------------------------------------------------------------------

void enterStochData (WitRun * theWitRun)
   {
   int periodStage[] = {0, 1};

   witSetNScenarios    (theWitRun, 3);
   witSetStageByObject (theWitRun, WitFALSE);

   witSetStochMode     (theWitRun, WitTRUE);

   witSetPeriodStage   (theWitRun, periodStage);

   enterScenarioData   (theWitRun, 0, 0.25, 200.0);
   enterScenarioData   (theWitRun, 1, 0.50, 300.0);
   enterScenarioData   (theWitRun, 2, 0.25, 400.0);
   }

//------------------------------------------------------------------------------

void enterScenarioData (
      WitRun * theWitRun,
      int      theScen,
      float    probability,
      float    demandVolPer1)
   {
   float demandVol[2];

   demandVol[0] = 0.0;
   demandVol[1] = demandVolPer1;

   witSetCurrentScenario (theWitRun,                 theScen);
   witSetProbability     (theWitRun,                 probability);
   witSetDemandDemandVol (theWitRun, "Hold", "Sell", demandVol);
   }
```

```
//------------------------------------------------------------------------------

void displaySoln (WitRun * theWitRun)
   {
   int     theScen;
   float * execVol;
   float * demandVol;
   float * shipVol;
   float   objValue;
   float   boundsValue;

   witGetOperationExecVol (theWitRun, "Buy",  & execVol);

   std::cout
      << "Buy "
      << execVol[0]
      << " papers."
      << std::endl
      << std::endl;

   witFree (execVol);

   for (theScen = 0; theScen < 3; theScen ++)
      {
      witSetCurrentScenario (theWitRun,                   theScen);

      witGetDemandDemandVol (theWitRun, "Hold", "Sell", & demandVol);
      witGetDemandShipVol   (theWitRun, "Hold", "Sell", & shipVol);

      std::cout
         << "If the demand is for "
         << demandVol[1]
         << " papers, sell "
         << shipVol[1]
         << " papers."
         << std::endl;

      witFree (demandVol);
      witFree (shipVol);
      }

   witGetObjValues (theWitRun, & objValue, & boundsValue);

   std::cout
      << std::endl
      << "The expected profit is $"
      << objValue
      << "."
      << std::endl;
   }
```

When this program is run, it produces the following output:

```
Buy 300 papers.

If the demand is for 200 papers, sell 200 papers.
If the demand is for 300 papers, sell 300 papers.
If the demand is for 400 papers, sell 300 papers.

The expected profit is $95.
```

APPENDIX B

File Formats

Input Data File

This section defines the format of the Input Data file, i.e., the file read by the API function `witReadData` and the main input file for the stand-alone executable.

The Input Data file is in “free format”: it consists of a series of “tokens” separated by “white space” (blanks and line breaks). The file format can be thought of as a “language” and the formal syntax for this language will be given below. But we begin with an informal description, including explanations of what the language is telling WIT to do.

A valid input data file consists of zero or more “commands”. The valid commands are:

- “add” command
- “set” command
- “read” command

Each command ends with a semicolon.

An “add” command tells WIT to create a new WIT data object (of some type) and to assign values to zero or more of the attributes associated with that object. Any attribute whose value is not assigned will retain its default value (defined in Chapter 2). An example of an “add” command is as follows:

```
add bomEntry "operation17" "material24"
   earliestPeriod 2
   latestPeriod   7;
```

A “set” command identifies an already existing object and tells WIT to assign values to zero or more of the attributes associated with that object. Any attribute whose value is not assigned will retain its current value. An example of a “set” command is as follows:

```
set demand "part43" "customer5"
   buildAheadLimit 4
   shipLateLimit   1;
```

A “read” command tells WIT to temporarily interrupt reading the current input data file, read a different input data file, and then resume reading the current file. An example of a “read” command is as follows:

```
read "supply.data";
```

The language also allows for comments. A comment begins with double slash (//), ends at the end of the line and may be inserted anywhere in the file, e.g.:

```
set demand "part43" "customer5"
   buildAheadLimit 4   // This is a comment.
   shipLateLimit   1;
```

Data Types

An attribute value may be any of the following types:

1. INTEGER
2. FLOAT
3. <boolean>
4. STRING
5. <vector_format>
6. <bound_set_format>

The format for INTEGER and FLOAT is just the usual format used in C and other languages.

A <boolean> is either true or false.

A STRING is any sequence of characters enclosed in double quote marks ("), with the following exceptions:

```
    "
```

is represented as:

```
    \"
```

and

```
    \
```

is represented as:

```
    \\
```

Thus the following string:

```
     abc\def"ghi
```

would be represented as:

```
    "abc\\def\"ghi"
```

A \ followed by anything other than " or \ is an error.

For convenience, three different formats are allowed for vectors:

- dense

  Each element of the vector must be specified. For example, if nPeriods = 4, the following is a dense vector format:

  ```
  dense (9. 8. 7. 6.)
  ```

- single

  One value is specified and every element of the vector is assigned this value. For example:

  ```
  single (2.3)
  ```

  is equivalent to:

  ```
  dense (2.3 2.3 2.3 2.3)
  ```

- sparse

  The values of some elements are specified and the rest are assigned the default value for the attribute. For each element to be specified, list the element's period, followed by a colon, followed by the value of the element. For example, if the default is 0., then

  ```
  sparse (3:4.2 1:7.8)
  ```

  is equivalent to:

  ```
  dense (0. 7.8 0. 4.2)
  ```

A bound set attribute is specified by specifying zero or more of the bounds that define it: hardLB, softLB, and hardUB. Each bound is specified as a vector. Any bound not specified in a bound set retains either its default value (in an "add" command) or its current value (in a "set" command). A bound set format is terminated with the "endBounds" keyword. For example:

```
set part "material56"
   stockBounds
      softLB sparse (2:10.)
      hardUB single (100.)
      endBounds;
```

In this case, the hardLB for the stockBounds of part ”material56” is left at its current value.

Formal Syntax

Following is a list of BNF (Backus-Nauer Form) rules which formally describe the syntax of the WIT input data file language. The terms enclosed in <> are defined by this syntax. The terms in non-proportional font (e.g., `add` or `part`) are literal and must be written exactly as shown. The terms in upper case (e.g., STRING, INTEGER) are left undefined in the syntax and are explained elsewhere.

<input_data> ::

    <command_list>
    | <release_specification> <command_list>

<release_specification> ::

    `release` <release_num>;

<release_num> ::

    STRING

<command_list> ::

    <empty>
    | <command_list> <command>

<command> ::

    <add_command>
    | <set_command>
    | <read_command>

<add_command> ::

    `add` <addable_object_type> <argument_list> <attribute_list>;

<set_command> ::

    `set` <settable_object_type> <argument_list> <attribute_list>;

<addable_object_type> ::

    <basic_object_type>

```
    | partWithOperation

<settable_object_type> ::
        <basic_object_type>
    | problem

<basic_object_type> ::
      part
    | demand
    | operation
    | bomEntry
    | subEntry
    | bopEntry

<argument_list> ::
      <empty>
    | <argument_list> <argument>

<argument> ::
      STRING
    | INTEGER
    | <part_category>

<part_category> ::
      material
    | capacity

<attribute_list> ::
      <empty>
    | <attribute_list> <attribute>

<attribute> ::
    ATTRIBUTE_NAME <attribute_value>
```

```
<attribute_value> ::
      <simple_value>
    | <vector_format>
    | <bound_set_format>


<simple_value> ::
      INTEGER
    | FLOAT
    | STRING
    | <boolean>


<boolean> ::
      true
    | false


<vector_format> ::
        dense (<value_list>)
    |   single (<vector_value>)
    |   sparse (<sparse_list>)


<value_list> ::
      <vector_value>
    | <value_list> <vector_value>

<vector_value> ::
      INTEGER
    | FLOAT


<sparse_list> ::
      <empty>
    | <sparse_list> <period_value>


<period_value> ::
        <period> : <vector_value>
```

<period> ::

    INTEGER

<bound_set_format>::

```
  endBounds
```
| <bound_item> <bound_set_format>

<bound_item> ::

    <bound_type> <vector_format>

<bound_type> ::

```
   hardLB
 | softLB
 | hardUB
```

<read_command> ::

    `read` <file_name>;

<file_name> ::

    STRING

Additional Language Rules

In addition to the above syntax, the following rules apply:

1. Note that the file may optionally begin with a release specification. The release specification doesn't actually mean anything; it is included in the present file format simply for compatibility with previous versions of the format.
2. An ATTRIBUTE_NAME can be the name of any input attribute listed in chapter 2, except for the following:
   - partName
   - demandName
   - operationName
   - appData
   - fssShipVol
   - optInitMethod

- solverLogFileName
- Any attribute listed as an "immutable input attribute" in Chapter 2

3. The ATTRIBUTE_NAME must match the object type being set or added, e.g., supplyVol is only allowed for a part.

4. In an "add partWithOperation" command, only part attributes may be specified. (The operation and bopEntry attributes for a partWithOperation can be set using the "set operation" and "set bopEntry" commands.)

5. The type and value of an <attribute_value> must be appropriate to the attribute, as defined in Chapter 2. E.g., execPenalty must be a float ≥ 0.0.

6. The number of <vector_value>s specified in a `dense` <vector_format> must be equal to nPeriods.

7. A <period>, t, must satisfy $0 \le t < nPeriods$.

8. The same <period> cannot be specified more than once within the same `sparse` <vector_format>.

9. A <bound_type> (i.e., hardLB, softLB, hardUB) must not be specified more than once within the same <bound_set_format>.

10. The <argument_list> in an "add" or "set" command is required to be the specific list of arguments appropriate for the object type being added or set. The specific arguments required for each "add" and "set" command is given in the table below. In all cases, the arguments are attributes of the corresponding object type. The types of these arguments/attributes are given in Chapter 2. Note that, in some cases, the arguments for an "add" command are somewhat different than the arguments for a "set" command for the same object type (e.g. consumedPartName vs. bomEntryIndex for a bomEntry). This simply reflects the fact that, in some cases, different information is needed to create an object than is needed to look up an existing object.

11. The STRING given as a <file_name> will be interpreted as the name of an input data file.

12. The read commands can be nested, e.g., file A contains a read command for file B, which contains a read command for file C, etc. There is a limit on the number of nested reads that are allowed. Currently the limit is 30.

13. The maximum number of characters allowed in a line of input is 1000. If a line in an input data file exceeds this limit, an error message is issued, and the program is terminated. This error message, which originates in the input reading software ("LEX") used by WIT, is **not** controlled by the "API Message Attributes". (See page 156.)

TABLE 9 **Arguments for the "add" and "set" Commands**

Object Type	Arguments for "add"	Arguments for "set"
part	partName partCategory	partName
demand	demandedPartName demandName	demandedPartName demandName

TABLE 9 Arguments for the “add” and “set” Commands

Object Type	Arguments for “add”	Arguments for “set”
operation	operationName	operationName
bomEntry	consumingOperationName consumedPartName	consumingOperationName bomEntryIndex
subEntry	consumingOperationName bomEntryIndex consumedPartName	consumingOperationName bomEntryIndex subsBomEntryIndex
bopEntry	producingOperationName producedPartName	producingOperationName bopEntryIndex
problem	“add” not allowed	no arguments
partWithOperation	partName	“set” not allowed

14. The object identified in a “set” command must already exist.

15. If an argument in a command identifies another object (e.g., consumedPartName), that object must already exist.

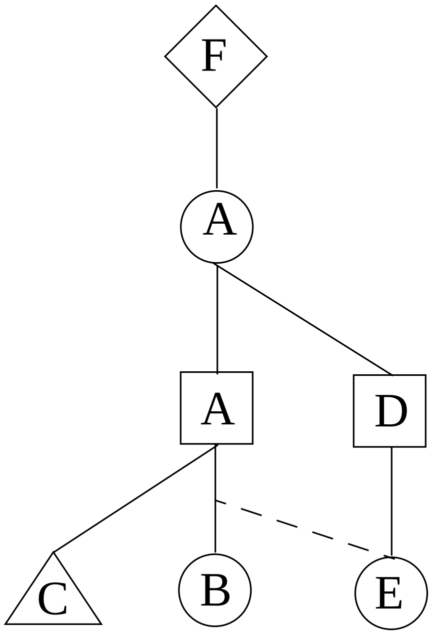
16. The various constraints on objects and attributes defined in Chapter 2 all apply, e.g., a part’s partName must be unique.

NOTE: Some of WIT’s error messages for the input data file refer to “entities”. “Entity” is a synonym for “object type”.

Sample Input Data Files

We conclude this section with an example pair of input data files which, together, define a small WIT problem. The example was designed to illustrate the various language constructs that the file format permits and is not a particularly meaningful WIT problem. The problem has the following structure:

FIGURE 22 The Problem Defined by the Example WIT Input Data Files



Here are the example files:

File “example.main.data”:

```
//------------------------------------------------------
// Example of a WIT Input Data File
// Main File
//------------------------------------------------------

//------------------------------------------------------
// Setting problem attributes.
//------------------------------------------------------

set problem
   nPeriods 4
   execEmptyBom true
   title "quote mark: \" back slash: \\";
     //
     // title: -->quote mark: " back slash: \<--

//------------------------------------------------------
// Creating Objects.
//------------------------------------------------------
```

```
      // Creates part A, operation A,
      // and the BOP entry connecting them.
      //
add partWithOperation "A"
   stockCost single (5.0)   // attributes of part A
   scrapCost single (50.0);

add part "B" material
   stockCost single (1.0)
   scrapCost single (10.0);

add part "C" capacity
   supplyVol single (30.0);

add part "E" material
   stockCost single (1.0)
   scrapCost single (10.0);


add demand "A" "F";


add operation "D";


add bomEntry "A" "C";

add bomEntry "A" "B";

add bomEntry "D" "E";


add subEntry "A" 1 "E";
      //
      // Substitution of part E in place of part B
      // in the BOM entry representing the consumption
      // of part B by operation A.
      // This BOM entry has bomEntryIndex = 1.


add bopEntry "D" "A";

//--------------------------------------------------------
// Reading supply data from another file
//--------------------------------------------------------

read "example.supply.data";

//--------------------------------------------------------
// Setting object attributes.
//--------------------------------------------------------
```

```
set part "A"
   stockBounds
      softLB single (10.0)
      hardUB dense (30.0 30.0 20.0 20.0)
      endBounds
   supplyVol sparse (0:42.0);
      //
      // Overrides the value, 17.0, given in
      // example.supply.data.

set demand "A" "F"
   demandVol dense (50.0 60.0 70.0 80.0)
   shipReward single (1000.0)
   cumShipReward single (10.0);

set operation "A"
   yieldRate single (0.95);

set bomEntry "A" 1
   offset single (1.0);

set subEntry "A" 1 0
   latestPeriod 2;

set bopEntry "A" 0
   productRate single (2);
```

File “example.supply.data”:

```
//--------------------------------------------------------
// Example of a WIT Input Data File
// Supply File
//
// This file specifies supply data for the parts
// defined in "example.main.data".
//--------------------------------------------------------

set part "A"
   supplyVol sparse (0:17.0);
      //
      // Initial inventory = 17.0.

set part "B"
   supplyVol sparse (1:100.0 3:50.0);

set part "C"
   supplyVol single (34.0);
      //
      // Overrides the value, 30.0, given in
      // example.main.data.

set part "E"
   supplyVol single (25.0);
```

Control Parameter File

The Control Parameter file is used only by the WIT stand-alone executable. It specifies run-time control parameters to WIT. Its default name can be overridden by specifying the file name as the command line option to WIT.

Format

- All character data is considered to be case-sensitive.
- In general, the data is read in free format, i.e., WIT reads in the file as a series of tokens where each token is separated by one or more blank spaces and/or one or more line breaks.

Control Parameter Defaults

Each control parameter has a name, a type, and default value. To specify the value of a control parameter, enter its name, followed by the value. Any parameter not specified in this file stays at its default.

Consider the following example:

```
print_echo yes
action heur
```

This Control Parameter file tells WIT to print an echo of the input and perform a Heuristic implosion using default values for all parameters not listed.

TABLE 10 **Control Parameters and their Defaults**

Name	**Type**	**Default**
data_ifname	String	Platform Dependent Default Information. See the table for your platform in the Preface section of this book
log_ofname	String	
echo_ofname	String	
pre_ofname	String	
solver_ofname	String	
soln_ofname	String	
exec_ofname	String	
ship_ofname	String	
mrpsup_ofname	String	
critical_ofname	String	
print_echo	yes/no	no
print_pre	yes/no	no
print_exec	yes/no	yes
print_ship	yes/no	yes
print_soln	yes/no	no
action	String	opt
auto_pri	yes/no	no
n_critical	integer	0
outputPrecision	integer	3
equitability	integer	1

Control Parameter Definitions

Control parameters have the following meaning:

Control Parameter	Meaning
data_ifname	Name of Input Data File
log_ofname	Name of the Status Log File
echo_ofname	Name of the Echo Output File
pre_ofname	Name of the Pre-processing Output File
solver_ofname	Name of the LP Solver Log File
soln_ofname	Name of the Comprehensive Implosion Solution Output File or of the Comprehensive MRP Solution Output File. Note that only one of these two files will ever be printed by any run of the stand-alone executable. See the “print_soln” control parameter.
exec_ofname	Name of the Execution Schedule Output File
ship_ofname	Name of the Shipment Schedule Output File
mrpsup_ofname	Name of the Requirements Schedule Output File
critical_ofname	Name of the Critical Parts List Output File
print_echo	Print the Echo Output File (yes/no)
print_pre	Print the Pre-Processing Output File (yes/no)
print_exec	Print the Execution Schedule (yes/no). The parameter only applies if parameter “action” is either “opt” or “heur”.
print_ship	Print the Shipment Schedule (yes/no). The parameter only applies if parameter “action” is either “opt” or “heur”.
print_soln	Print either the Comprehensive Implosion Solution Output File or the Comprehensive MRP Solution File (yes/no). If yes is specified, and the “action” control parameter is “opt”, “heur”, or “preproc”, then the Comprehensive Implosion Solution Output File will be printed. In this case,

	a Focussed Shortage Schedule with universal focus is also computed and printed as part of the Comprehensive Implosion Solution. If yes is specified, and the “action” control parameter is “mrp”, then the Comprehensive MRP Solution Output File will be printed.
action	Tells which action is to be performed. There are four possible values for this parameter:

Value	Meaning
heur	Performs heuristic implosion
opt	Performs optimizing implosion.
stoch	Performs stochastic implosion.
mrp	Performs WIT-MRP.
preproc	Preprocessing only (Useful for debugging input. Note: performs preprocessing for optimizing implosion.)

auto_pri	This parameter invokes the automatic priority feature of WIT. If the above parameter action definition is heur or preproc, and auto_pri is yes, then WIT ignores the priorities given in the input and generates its own priorities for the Heuristic, based on objective function data. In this case, the Objective Choice must be 1 or 2. (See “objItrState” on page 108.)
	If parameter action is opt, then this parameter has no effect. (In the opt case, priorities for the Heuristic are generated automatically, regardless of the setting of auto_pri.)
n_critical	This parameter is only meaningful if the parameter action is either heur or opt. If n_critical is not 0, this indicates that, after an implosion is performed, a Critical Parts List is to be generated and printed.

	For more information about the Critical Parts List, see “Additional Capabilities of WIT” on page 32. If n_critical < 0, the entire Critical Parts List will be printed. If n_critical > 0, only the first n_critical parts and periods will be printed.
outputPrecision	This parameter is the value of the “outputPrecision” attribute defined in chapter 2. It must be an integer ≥ 0.
equitability	This parameter is the value of the “equitability” attribute (for equitable allocation) defined in chapter 2. This parameter must be an integer, $1 \leq$ equitability ≤ 100.

Output Files

WIT output files (the Execution Schedule, Shipment Schedule, Requirements Schedule, and Critical Parts List) have the following format features in common:

- A field width of 14 (including the quotation marks) is used for any name length less than or equal to 12 characters. Any name of with a length greater than 12 characters forces the field width to be expanded to length+2 for that name only.

- WIT works with period numbers, e.g., in a 26 period problem, the period numbers are 0 to 25.

- With the exception of the Critical Parts List, parts are printed in the order in which WIT internally stores them. This might not match the order in which the parts were entered.

Each line begins with an optional message number, which can be turned off (made not to print) using `witSetMesgPrintNumber` in API mode. The message numbers do not print in stand-alone mode are not shown in the formats given in this guide.

Execution Schedule Output File

This file displays the Execution Schedule, one of the main results of implosion. This file consists of either one or two sections, depending whether or not substitutes are present in the data. Each line of the first section is in the following column format:

TABLE 11 **Execution Schedule Output File Part I Format**

Data	Field Width	Type
operationName	14 or more	Quoted string, left justified.
Blank	2	
Completion Period	3	Right-justified integer ≥ 0
Blank	2	
execVol for this operation. For a definition see “execVol” on page 132.	11	Float > 0.0, right-justified and normally printed to 3 decimal places.

The number of decimal places to which the execVol field is printed is given by the outputPrecision attribute, which defaults to 3.

For each operation, there is one line for each period in which execVol is positive and no line for any period in which execVol is zero.

Consider the following example: (one line in the above format)

```
"Opn1"          12      654.000
```

For each operation, the `execVols` are printed in order of increasing period. This line indicates that 654 units of operation “Opn1” are to be executed in period 12.

A second section of this file is printed only if there are substitutes. For each substitute BOM entry, it gives the execution volumes due to that substitute.

After a header line for this section, the following column format is used:

TABLE 12 **Execution Schedule Output File Part II Format**

Data	**Field Width**	**Type**
consumingOperationName	14 or more	Non-blank string. Trailing blanks
Blank	2	
ReplacedPartName	14 or more	Non-blank string. Trailing blanks
Blank	2	
bomEntryIndex	8	Right Justified integer ≥ 0
Blank	2	
Period Number	3	Right Justified integer ≥ 0
Blank	2	
consumedPartName	14 or more	Non-blank string. Trailing blanks
Blank	2	
subsBomEntry Index	8	Right Justified integer ≥ 0
Blank	2	
subVol for this substitute. For a definition see “subVol” on page 147	11	Right justified float > 0.0 normally printed to 3 decimal places.

The number of decimal places to which the subVol field is printed is given by the outputPrecision attribute, which defaults to 3.

For each substitute BOM entry, there is one line for each period in which subVol is positive and no line for any period in which subVol is zero.

This section is ordered by:

- Operation
- BOM entry
- Period
- Substitute BOM entry

Consider the following example:

Ignoring column numbers, an example of two lines in this format is:

```
"Opn1" "Comp1" 4 12  "Comp2"   1    24.000
"Opn1" "Comp1" 4 12  "Comp4"   3    51.000
```

The first line indicates that 24 units of operation Opn1 are to be executed in period 12 using component Comp2 in place of component Comp1. The Comp1 is BOM entry index 4 in the BOM of operation Opn1 and Comp2 is substitute BOM entry number 1.

The second line indicates that 51 units of Opn1 are to be executed in period 12 using component Comp4 in place of component Comp1. The Comp1 is BOM entry index 4 in the BOM of operation Opn1 and Comp4 is substitute BOM entry number 3. If the Execution Schedule indicates that 654 units of operation Opn1 are to be executed in period 12, then the remaining 579 units are executed using component Comp1, without substitution.

Shipment Schedule Output File

This file displays the Shipment Schedule, one of the main results of implosion. Each line of this file is in the following column format:

TABLE 13 **Shipment Schedule Output File Format**

Data	Field Width	Type
demandedPartName	14 or more	Quoted string, left justified.
Blank	2	
demandName	14 or more	Quoted string, left justified.
Blank	2	
Period number	3	Right-justified integer ≥ 0
Blank	2	
shipVol For a definition see “shipVol” on page 129.	14	Float > 0.0, right-justified and normally printed to 3 decimal places.

The number of decimal places to which the shipVol field is printed is given by the outputPrecision attribute, which defaults to 3.

For each demand stream, there is one line for each period in which the shipVol is positive, and no line for any period in which the shipVol is zero. If there is any demand stream for which WIT allocates no shipment, that demand stream will not appear in this file at all.

For each demand stream, the shipVols are printed in order of increasing period. The shipVol is given in floating point format. (The shipVols are printed as floating point numbers for compatibility with other WIT file formats.)

Consider the following example:

Ignoring column numbers, an example of 5 lines in this format is

```
"Part1"   "P1Dem1"     2        159.123
"Part1"   "P1Dem1"     3         43.456
"Part1"   "P1Dem2"     0         85.789
"Part2"   "P2Dem2"     2          9.012
"Part2"   "P2Dem2"     4          3.345
```

Assuming this is a 5 period problem, this output shows that the allocated shipments of part Part1 to demand P1Dem1 are:

- 0.0 in period 0
- 0.0 in period 1
- 159.123 in period 2
- 43.456 in period 3
- 0.0 in period 4

Also, if there was a demand stream P2Dem1 for part Part2, then from its absence in the output you can assume there were no shipments allocated to it.

Requirements Schedule Output File

This file displays the Requirements Schedule, the main result of WIT-MRP. Each line of this file is in the following column format:

TABLE 14 **Requirements Schedule Output File Format**

Data	Field Width	Type
partName	14 or more	Quoted string, left justified.
Blank	2	
Period number		Right-justified integer ≥ 0
Blank	2	
reqVol For a definition see "reqVol" on page 123.	11	Float > 0.0, right-justified and printed to 3 decimal places.

For each part there is one line for each period in which the required supply (as determined by WIT-MRP) is positive, and no line for any period in which the amount is zero. If there is any part for which the `reqVol` are all zero, then that part will not appear in this file at all.

For each part, the `reqVol` are printed in order of increasing period. The `reqVol` is rounded up to the nearest integer. Volumes are printed as floating point numbers for compatibility with other WIT file formats.

Consider the following example:

```
"Part1"   0    85.000
"Part1"   2   159.000
"Part1"   3    43.000
"Part2"   2     9.000
"Part2"   4     3.000
```

Assuming this is a 5 period problem, one can conclude that the required supply volumes of Part1 are:

- 85.000 in period 0
- 0.0 in period 1
- 159.000 in period 2
- 43.000 in period 3
- 0.0 in period 4

Critical Parts List Output File

This file contains the optional Critical Parts List. See “Critical Parts List” on page 38 for an explanation of this feature. Each line of this file is in the following column format:

TABLE 15 Critical Parts List Output File Format

Data	Field Width	Type
partName	14 or more	Quoted string, left justified.
Blank	2	
Period number	3	Right-justified integer ≥ 0

Consider the following example: (the first three lines of this file)

```
"Part17"          1
"Part17"          0
"Part08"          4
```

The list is ordered from the most critical part to the least critical. The above example suggests that the best candidate for improving the solution would be increasing the supply of Part17 in period 1, the second best candidate would be Part17 in period 0, and the third best candidate would be Part08 in period 4.

APPENDIX C References

1. Birge, J. 1997. “Stochastic Programming Computation and Applications”, INFORMS Journal on Computing Vol. 9, No. 2, pp. 111-133.
2. Wittrock, R. 2006. “An Introduction to WIT: Watson Implosion Technology”, Research Report RC24013; also available at:

   `http://domino.research.ibm.com/library/`
   `cyberdig.nsf/papers?SearchView&Query=RC24013`
3. Wittrock, R. 2006. “WIT’s Mathematical Programming Formulation of the Implosion Model”. Available from the author.

"They reel to-and-fro, and stagger like a drunken man: and are at their wit's end."

Prayer Book (1662)