IBM Watson Implosion Technology Release 6.0

User’s Guide and Reference

June 2, 2005 – Draft Document

Second Edition (March 2000)

Property of IBM

PREFACE

About this Book

Watson Implosion Technology (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. In addition, there is a WIT 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. If you are reading this guide just to get a clear idea of WIT's capabilities, we recommend reading Chapter 1.

How this Book is Organized

Chapter 1–Introducing WIT: Watson Implosion Technology

- 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

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
OSL	IBM Optimization Subroutine Library
WIT	Watson Implosion Technology
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 following Technical Descriptions are for advanced WIT users.

The information that appears between these two diamond symbols is for experienced WIT users and may be skipped during a first reading.

This concludes the Technical Descriptions for advanced WIT users.

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

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM’s product, program, or service may be used. Any functionally equivalent product, program, or services that does not infringe any of IBM’s intellectual property rights or other legally protectible rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, programs, or services, except those expressly designated by IBM, are the user's responsibility.

IBM may have patents or pending patent application, covering subject matter in this book. This furnishing of this book does not give you any license to these patents.

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)

Platforms Supported

Refer to IBM's marketing literature for the full list of supported platforms.

NOTE: Most of the information in this book is not platform specific. However, examples are primarily for RS/6000 users.

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
OSL Log File	osl.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`
- `witGetPipSeq`
- `witSetBopEntryPipShare`
- `witGetBopEntryPipShare`
- `witBuildPip`
- `witGetPipExists`
- `witGetDemandExecVolPip`
- `witGetDemandSubVolPip`
- `witGetDemandSupplyVolPip`
- `witGetDemandProdVolPip`
- `witGetDemandConsVolPip`
- `witGetDemandSideVolPip`

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 applied 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.

CHAPTER 1 Introducing WIT:Watson Implosion Technology

"His foe was folly and his weapon wit."

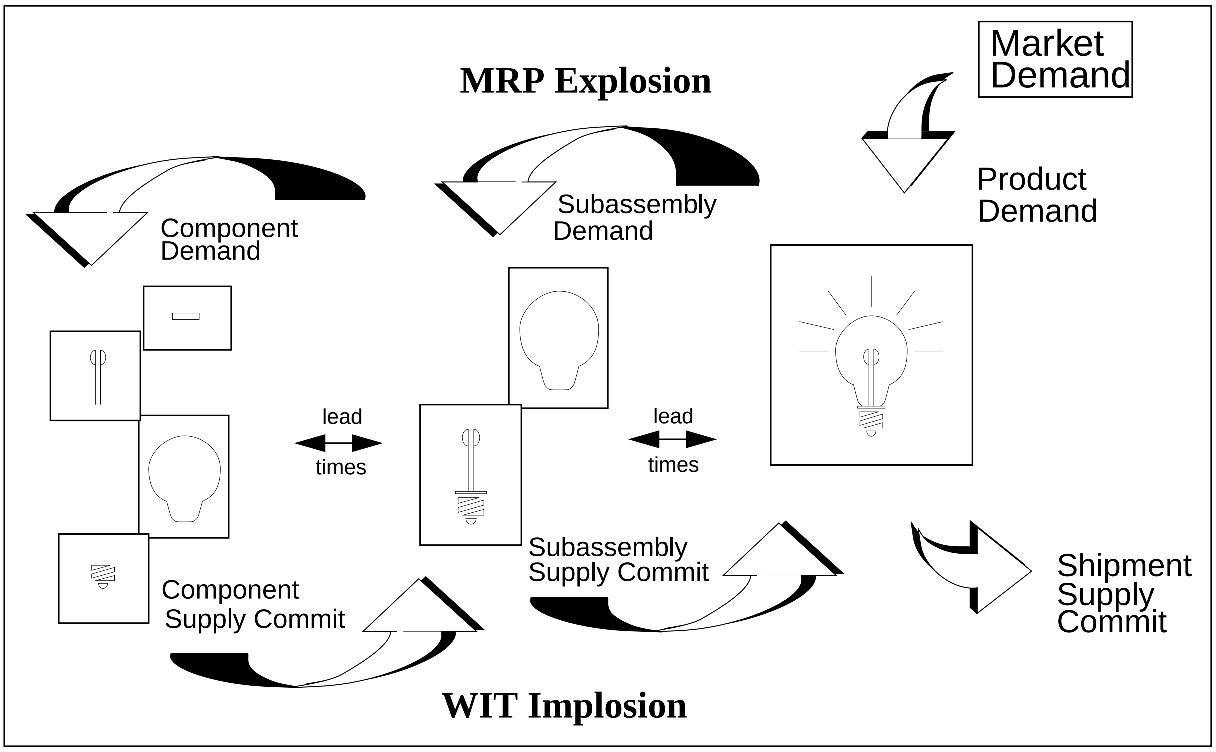
Anthony Hope
Inscription on the tablet to W.S. Gilbert
Victoria Embankment, London
1915

What is WIT?

Watson Implosion Technology (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.

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.

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

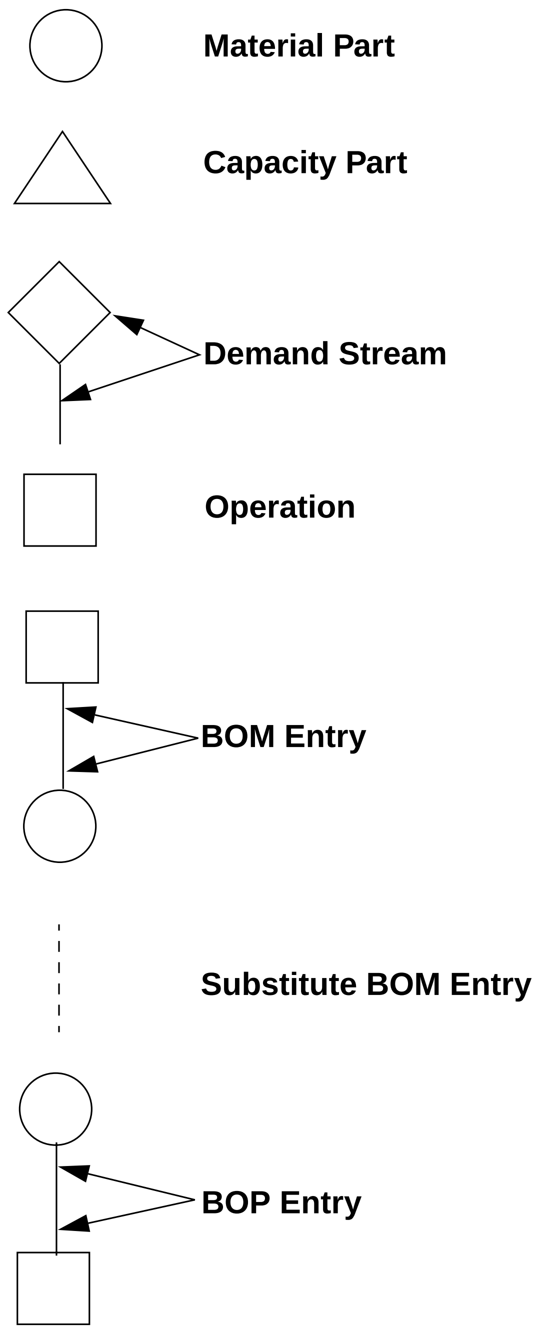
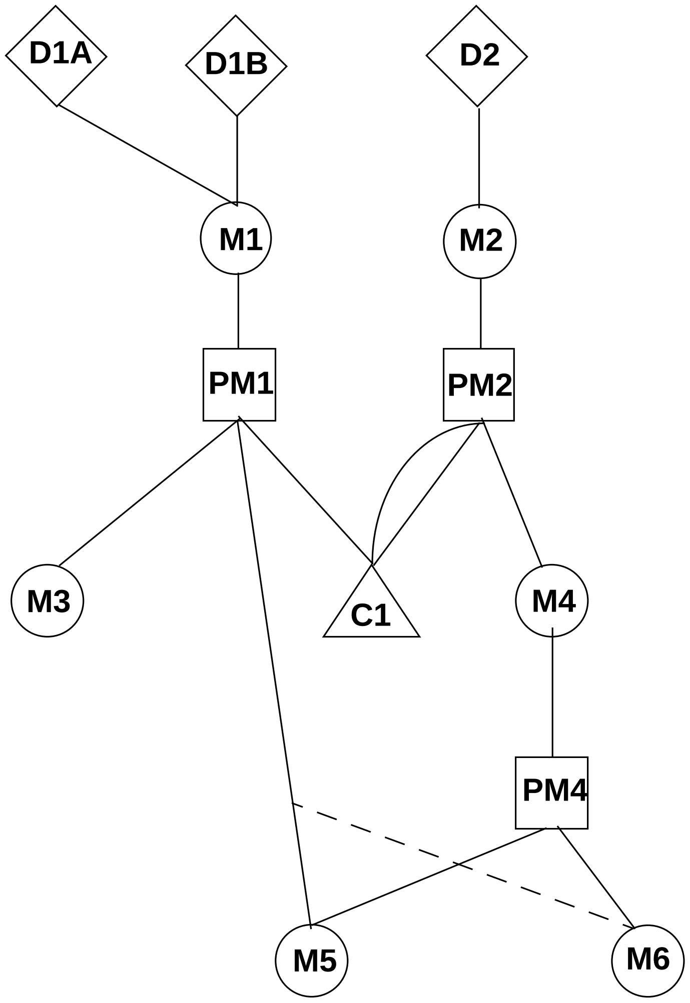


Figure 3 illustrates an example of an SMA problem.

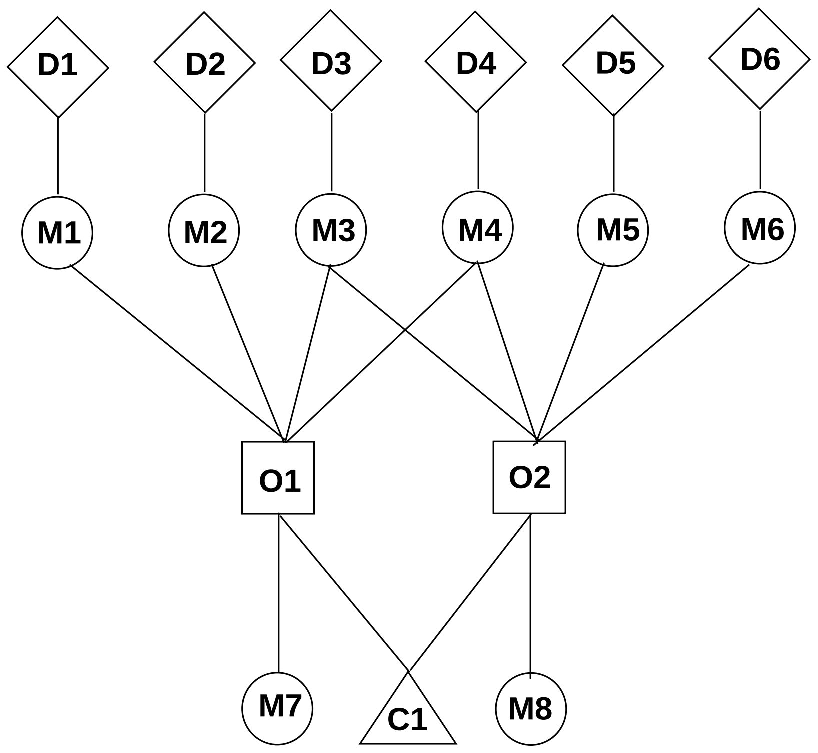
FIGURE 3 A Single Method Assembly Problem



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.

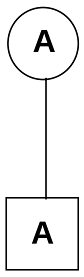
NOTE: The use of the word “part” to describe capacity may at first seem rather surprising. It’s clear that a material part has some rather different properties from, say, an hour of machine time. Internally, WIT carefully models all of the essential differences between material parts and capacities. 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 performs its implosion by using one of two completely different methods, at the user’s discretion:

- Optimizing implosion, or
- Heuristic implosion

The distinction between these two 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. Optimizing implosion finds the “best possible” implosion solution, according to an objective function that defines the “goodness” of any possible solution. Heuristic implosion finds a feasible implosion solution, guided by priorities set by the user, but without any measure of “goodness”.

Another difference between the two methods is that heuristic implosion runs much faster and requires less memory than optimizing implosion.

Some properties common to heuristic implosion and optimizing implosion 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. However, since implosion uses different logic than MRP explosion, it is possible that an MRP explosion of the Shipment Schedule may indicate shortages.
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`.

What Heuristic Implosion Does

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 63.) 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 59. 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 59.
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 91. 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 51.

What Optimizing Implosion Does

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 function that takes an implosion solution as its input and converts that into a single number that defines the goodness of the implosion solution. In fact, WIT has been designed to work with either of two objective functions (at the user's choice):

Objective #1:	A completely general objective.
Objective #2:	A relatively user-friendly objective.

Optimizing implosion 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 OSL which is an IBM* program product for solving optimization problems. The solution to the LP is then translated into an implosion solution.

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 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. With objective #1, optimizing implosion is completely well suited for solving non-SMA problems.
9. With objective #2, optimizing implosion will find an optimal solution for a non-SMA problem, but the objective function may not be a completely appropriate model.
10. Optimizing implosion is not allowed, if the problem contains no parts. (This restriction is actually applied during preprocessing for optimizing implosion.)
11. At the user’s request, optimizing implosion will compute “shadow prices” for the parts. See “shadowPrice” on page 100 and “compPrices” on page 79.

Optimizing Implosion Objective #2

Since Objective #1 is more advanced than Objective #2, Objective #2 will be described first, followed by Objective #1.

The Essential Objective #2: Overview

We begin our discussion with an overview of an “essential” version of Objective #2, which assumes that the implosion problem includes no substitutes and no soft lower bounds. This overview is followed by a detailed description of the essential version of Objective #2. Lastly we discuss the (non-essential) extensions to Objective #2 for handling substitutes and soft lower bounds.

The essential Objective #2 is an economic objective. It is computed from two main kinds of economic data from the user:

- unitCost: The standard unit cost of each part.
- grossRev: The per-unit gross sales revenue for each demand stream.

The essential Objective #2 is a composite of three economic objectives:

- obj2RevValue: Total Net Revenue.

  Approximately, this is the grossRev of all parts shipped minus their unitCost.
- obj2InvValue: Average Inventory.

  The in-process and finished goods inventory, measured in monetary terms and averaged over the time horizon.
- obj2ServValue: Aggregate Serviceability.

  A weighted average the serviceabilities of all demand streams in all periods, where the serviceability of a particular demand stream in a particular period is defined to be the fraction of the demand stream’s cumulative demand volume that has been met by the end of that period.

Having defined these three individual objectives, the essential Objective #2 is computed as a composite (a weighted sum) of the three. Notice however that these three objectives are not directly comparable. For example, a $1 million improvement in inventory is not worth the same as a $1 million improvement in revenue. For this reason, WIT applies scaling factors to the revenue, inventory and serviceability objectives when combining them to form Objective #2. In addition to WIT’s scaling factors, the user is allowed to set the following weights on each of the objectives:

- obj2Wrev: The weight on obj2RevValue. It is specified as a float $\geq$ 0.0.The “normal” value is 1.0. Higher values give more importance to the revenue objective; lower values give less importance to it. A value of 0.0 will cause WIT to ignore obj2RevValue.
- obj2Winv: The weight on obj2InvValue. The comments on obj2Wrev apply here, too.
- obj2Wserv: The weight on obj2ServValue. The comments on obj2Wrev apply here too.

WIT’s scaling factors on the objectives has been designed so that the essential objective #2 will make economic sense if the user specifies the weights as:

obj2Wrev = obj2Winv = obj2Wserv = 1.0

In particular, when the above weights are used, the essential objective #2 is equivalent to maximizing profit.

The following Technical Descriptions are for advanced WIT users.

The Essential Objective #2: Detailed Description

In addition to the execution schedule and the shipment schedule, the essential Objective #2 uses the following data:

- For each part: unitCost.

  This is the standard unit cost of the part. If the part is a product the unitCost should include the cost of the materials and capacity that go into making the product plus any value-add costs (for example, labor). It should normally reflect the least expensive way to build the product.
- For each demand stream: grossRev.

  This is the gross sales revenue received for each unit shipped to the demand stream.
- For each operation: obj2AuxCost

  The “auxiliary cost” of executing the operation. If there is any cost in executing an operation not reflected in the unitCost of the parts it produces, such costs should be included in the operation’s obj2AuxCost.

Consider the following example:

There is a part, A, which is produced by two different operations, B, and C, which produce no parts other than A. The total cost of A is $80 if it is built using operation B, but $100 if using operation C. In this case, the unitCost of A should be given as $80, the obj2AuxCost of B as $0 and the obj2AuxCost of C as $20.

In addition, the following quantities are used:

• nPeriods:	The number of periods in the planning horizon.
• periodsPerYear:	The number of periods in a year.
• capCost:	The annual percentage interest rate to be used for most purposes.
• invCost:	The annual percentage interest rate to be used for inventory.
• obj2Wrev:	The weight on obj2RevValue. It is specified as a float $\geq$ 0.0. The “normal” value is 1.0. Higher values give more importance to the revenue objective; lower values give less importance to it. A value of 0.0 will cause WIT to ignore obj2RevValue.
• obj2Winv:	The weight on obj2InvValue. The comments on obj2Wrev apply here, too.

- obj2Wserv: The weight on obj2ServValue. The comments on obj2Wrev apply here, too.

Given this data, the essential Objective #2 is computed as a composite of the following 3 objectives:

- obj2RevValue = Total net revenue.

  As a preliminary step, for each demand stream, i, for part j, the net revenue is computed as:

  $\texttt{netRev}_i = \texttt{grossRev}_i - \texttt{unitCost}_j$

  If, due to anomalies in the data, this turns out to be a negative number, it is set to zero. obj2RevValue is computed as follows: for each period, t, and each demand stream, the level of planned shipment for this demand stream in period t (shipVol_t) is multiplied by the netRev of the demand stream. The main term of obj2RevValue is computed as the sum of this over all periods and demand streams. As a second term, the total auxiliary cost is computed as the sum over all operations and periods of the operation’s obj2AuxCost times the execution volume (execVol_t) and this sum is subtracted from the first term. Finally, if WIT is forced to scrap the inventory of a part, the unitCost of each unit scrapped (scrapVol_t) is charged against obj2RevValue. (WIT scraps all unused supply of “capacity” parts and scraps the inventory of a product in any period in which it undergoes a mandatory E/C.)

- obj2InvValue = The in-process and finished goods inventory, measured in monetary terms and averaged over the time horizon.

  This objective is computed as a main term and an adjustment. The term is computed as follows: For each period and each part, the number of units in inventory (stockVol_t) is multiplied by the unitCost of the part. The total inventory in one period, t, is computed as the sum of this over all parts. The main term is then computed as the total inventory averaged over all periods.

  This main term overlooks one aspect of inventory. Suppose an operation has BOM entries and/or BOP entries whose offsets are not 0. Then the time between the consumption of the parts consumed and the production of the parts produced is not accounted for in the main term of obj2InvValue. To compensate for this, an adjustment is made to obj2InvValue so that it accounts for the inventory of all the parts consumed by an operation from the periods in which they are consumed up until the period in which the operation is executed and then accounts for the inventory of all the parts produced from the period in which the operation is executed until the periods in which they are produced.
- obj2ServValue = Aggregate serviceability.

  For a given demand stream, i, in a given period t, the serviceability is defined as:

$$\texttt{service}_{i,t} = \texttt{cumShipVol}_{i,t} \ / \ \texttt{cumDemandVol}_{i,t}$$

cumShipVol$_{i,t}$ is the cumulative shipment volume to demand stream i in period t, i.e., the total number of units shipped in periods 0, 1, ..., t. Similarly, cumDemandVol$_{i,t}$ is the cumulative demand volume to demand stream i in period t, i.e., the total number of units demanded in periods 0, 1, ..., t. Note that service$_{i,t}$ is a number between 0.0 and 1.0, where 1.0 represents perfect serviceability.

obj2ServValue is computed as the following weighted average of all the individual serviceabilities.

$$\texttt{obj2ServValue} = \sum_{i,t} \texttt{serviceWeight}_{i,t} * \texttt{service}_{i,t}$$

where the above sum is over all demand streams, i, and periods, t. The weights serviceWeight$_{i,t}$ are defined below.

To understand serviceWeight$_{i,t}$, we must first define two related quantities, serviceScale$_{i,t}$ and totalServiceScale. For each demand stream, i, and period, t, let

$$\texttt{serviceScale}_{i,t} = \texttt{netRev}_{i} * \texttt{cumDemandVol}_{i,t}$$

This is the net revenue corresponding to complete satisfaction of cumulative demand. Next, let

$$\texttt{totalServiceScale} = \sum_{i,t} \texttt{serviceScale}_{i,t}$$

where the above sum is over all demand streams, i, and periods, t. Finally, for each demand stream, i, and period, t, let

$$\texttt{serviceWeight}_{i,t} = \texttt{serviceScale}_{i,t} \ / \ \texttt{totalServiceScale}$$

So what does all this mean? First, notice that serviceWeight$_{i,t}$ is defined in such a way that serviceWeight$_{i,t} \geq 0.0$ and

$$\sum_{i,t} \texttt{serviceWeight}_{i,t} = 1.0$$

This implies that obj2ServValue is indeed a weighted average of the serviceabilities of each demand stream in each period (because the weights, serviceWeight$_{i,t}$, are nonnegative and sum to 1.0).

The other point to notice is that serviceWeight$_{i,t}$ is defined to be proportional to serviceScale$_{i,t}$, which is the net revenue corresponding to complete satisfaction of cumulative demand for demand stream i in period t. This is an economically commensurate way of comparing the serviceability of different demand streams. For example, if one demand stream has $1 million worth of cumulative demand and another demand stream has $10 million worth of cumulative demand, then when obj2ServValue gets computed, the serviceability of the second demand stream gets 10 times more weight than the serviceability of the first demand stream, because serviceWeight$_{i,t}$ is 10 times larger for the second demand stream than for the first.

Having defined these three individual objectives, the essential Objective #2 is computed as a weighted sum of the three. As was mentioned in the overview, these three objectives are not directly comparable, and so WIT applies scaling factors to the revenue, inventory and serviceability objectives when combining them to form Objective #2. The following scaling factors are used:

• revFactor:	The scaling factor for obj2RevValue.
• invFactor:	The scaling factor for obj2InvValue.
• servFactor:	The scaling factor for obj2ServValue.

How these scaling factors are computed is discussed below. They are used to compute the essential Objective #2 as follows:

```
objValue = revFactor * obj2RevValue
           - invFactor * obj2InvValue
           + servFactor * obj2ServValue
```

Since optimizing implosion maximizes Objective #2, this will tend towards maximizing obj2RevValue, minimizing obj2InvValue, and maximizing obj2ServValue. However, since it is not usually possible to optimize more than one objective at the same time, this objective will make trade-offs between these three potentially conflicting objectives. How these trade-offs are made depends on the scaling factors.

The scaling factors are computed as follows:

- revFactor = obj2Wrev

  If the user has specified the normal value of 1.0 for obj2Wrev, the resulting revFactor is 1.0. This simply means that Objective #2 is being expressed units comparable to revenue.
- invFactor = obj2Winv * (nPeriods / periodsPerYear) * (invCost / 100)

  Suppose the user has specified the normal value of 1.0 for obj2Winv. Since obj2InvValue is the average inventory over the time horizon, and invCost is the annual percentage cost of inventory and (nPeriods / periodsPerYear) is the fraction of a year corresponding to the time horizon, it follows that invFactor * obj2InvValue

is the total cost of the inventory over the time horizon.

Consider the following example:

If:

```
   obj2RevValue   = $20,000,000
   obj2InvValue   = $10,000,000
   invCost        = 30
   nPeriods       = 26
   periodsPerYear = 52
   obj2Wrev       = 1.0
   obj2Winv       = 1.0
then
   invFactor = 1.0 * (26 / 52) * .30
             = 0.15
```

and the first two parts of the objective combine as:

```
   $20,000,000 - 0.15 * $10,000,000 = $18,500,000
```

I.e., $20 million worth of parts were sold, but an average of $10 million worth of material was held in inventory for half a year, at a cost of $1.5 million.

- servFactor =
  obj2Wserv * ((capCost / 100) / periodsPerYear) * totalServiceScale

  To see why this formula is was chosen for servFactor, assume that the user has specified the normal value of 1.0 for obj2Serv. The backlog volume for demand i in period t is given by the following formula:

  $\texttt{backlogVol}_{i,t} = \texttt{cumDemandVol}_{i,t} - \texttt{cumShipVol}_{i,t}$

  If you work out the algebra, it turns out that the serviceability term in objective #2 (i.e., servFactor * obj2ServValue) is implicitly assigning the following penalty to each unit of backlog for demand i in period t:

  $\texttt{backlogPenalty}_{i,t} =$
  $\texttt{((capCost / 100) / periodsPerYear) * netRev}_{i}$

  Keeping in mind that capCost is an annual percentage interest rate, one can see that the above formula is precisely one period’s worth of interest on the net revenue that would be received for demand i. Since each unit of backlog represents a delay of one period in receiving the revenue, this is the economically appropriate penalty for backlog. Our formula for servFactor was chosen simply to make this implicit formula for the backlog penalty work out correctly.

We mentioned in the overview that if the user specifies the weights as:

```
obj2Wrev = obj2Winv = obj2Wserv = 1.0
```

then the essential objective #2 is equivalent to maximizing profit. To see this, first notice that several aspects profit are accounted for in obj2RevValue: gross revenue (in netRev), the cost of materials and production (in the unitCost part of netRev), and the cost of materials and production wasted (in the scrapping charge to obj2RevValue). Next, the cost of inventory is accounted for in invFactor * obj2InvValue. Finally, the opportunity cost of delayed revenue is accounted for in servFactor * obj2ServValue (through its implicit penalty on backlog). We believe that these three terms that make up the essential objective #2 provide a reasonably good model of profit for optimization purposes in the sense that profit will be maximized when the essential objective #2 is maximized.

When Objective #2 is being used, WIT works in the following way: After the user provides the necessary data listed above, including weights (along with all the other implosion data), WIT finds the optimal value for Objective #2, objValue. It then reports back (along with the implosion solution) the value of objValue, but also the value of obj2RevValue, obj2InvValue and obj2ServValue.

The user can then run WIT repeatedly with different values of `obj2Wrev`, `obj2Winv` and `obj2Wserv` to see the effect on obj2RevValue, obj2InvValue, and obj2ServValue. In this way, the user is able to make trade-offs between revenue, inventory and serviceability.

We recommend starting with
`obj2Wrev = obj2Winv = obj2Wserv = 1.0`
for the first run and then adjusting from there. For example, if obj2InvValue is too large in the first run, increase `obj2Winv` for the second run, e.g. to 2.0.

<u>Recommendations for Objective #2:</u>

- If you specify a grossRev for a demand stream that is less than the unitCost of the part demanded, WIT will compute a netRev of 0. This is as if the grossRev were equal to the unitCost. While this is acceptable to WIT, it may lead to an undesirable implosion solution. In this case, WIT has no incentive to schedule any shipment to this demand stream, except for the purposes of reducing inventory. This may result in the shipments being less or later than they should be. Therefore, we recommend to always specify a grossRev greater than the unitCost of the part.
- Similarly, it is allowable to specify 0 as the unitCost of a material part, but this could also lead to an undesirable implosion solution. Since the inventory objective is based on the unitCost of each material part, WIT has no incentive to minimize the inventory of a part whose unitCost is zero. We recommend a positive unitCost be always specified for material parts.
- In most cases, the unitCosts of capacities should be given as 0. Since WIT considers each unit of unused capacity to be “scrapped”, the unitCost of all unused capacity is charged against obj2RevValue as a scrapping cost and so the unitCost becomes a penalty for not using this capacity. Note that this may cause WIT to build a product for which there is no demand, simply to avoid

scrapping capacity. We believe that penalizing the failure to use a capacity is usually inappropriate, but it may be useful in some cases, so positive unitCosts on capacities are allowed.

- We recommend specifying capCost, invCost, obj2Wrev, obj2Winv, and obj2Wserv all as strictly positive. Specifically, it is necessary to specify capCost and obj2Wserv as positive to provide an incentive to ship to demands on time. It is necessary to specify invCost and obj2Winv as positive to provide an incentive to avoid unneeded inventory. It is necessary to specify obj2Wrev as positive to provide an incentive to avoid scrapping.

Objective #2 with Substitutes

If the problem includes substitutes, Objective #2 is extended in the following two ways.

1. When one part is substituted for another part, the difference in the unitCost between the two parts is charged against the revenue objective.

Consider the following example:

Suppose that part B has a unitCost of $20 and appears in the BOM of operation A with a usage rate of 5. So, $100 worth of part B are normally consumed in the execution of operation A.

Now suppose part C has a unitCost of $21 and appears as a substitute for part B in the execution of operation A with a usage rate of 6. So, $126 worth of part C are consumed in the execution of operation A when this substitution is made. If 2 units of operation A are executed using part C instead of part B, the additional cost incurred by using part C instead of part B is:

2 * ($126 - $100) = $52

This $52 is charged against the revenue objective. Note that it is possible for this difference to be a negative number, in which case the revenue objective is increased.

2. In addition to the revenue adjustment, Objective #2 includes a substitute objective, obj2SubValue, that allows the user to penalize the use of substitutes. The input data for this objective is as follows:

obj2Wsub:	The weight on the substitute objective. It is specified as real number ≥ 0.0. The “normal” value is 1.0. Higher values give more importance to the substitute objective; lower values give less importance to it. A value of 0.0 will cause WIT to ignore the substitute objective (possibly a valid thing to do).

For each substitute BOM entry, a penalty is specified as a real number ≥ 0.0. This penalty is used to compute the substitute objective, obj2SubValue, as follows:

For each period and each substitute BOM entry, the execution volume due to the substitute BOM entry is multiplied by the penalty on the substitute. obj2SubValue is then defined as the sum of this over all periods and all substitute BOM entries.

NOTE: If the substitute is truly interchangeable with the part for which it is substituting, then the penalty should be specified as 0.0.

With substitutes, objective #2 is defined as follows:

```
objValue =  revFactor * obj2RevValue
          - invFactor * obj2InvValue
          + servFactor * obj2ServValue
          - obj2Wsub * obj2SubValue
```

where obj2RevValue is adjusted as mentioned above.

Objective #2 with Soft Lower Bounds

3. If the input data includes soft lower bounds (that are larger than the corresponding hard lower bounds), Objective #2 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 26.

With soft lower bounds and substitutes, objective #2 is defined as follows:

```
objValue = revFactor * obj2RevValue
     - invFactor * obj2InvValue
     + servFactor * obj2ServValue
     - obj2Wsub * obj2SubValue
     - wbounds * boundsValue
```

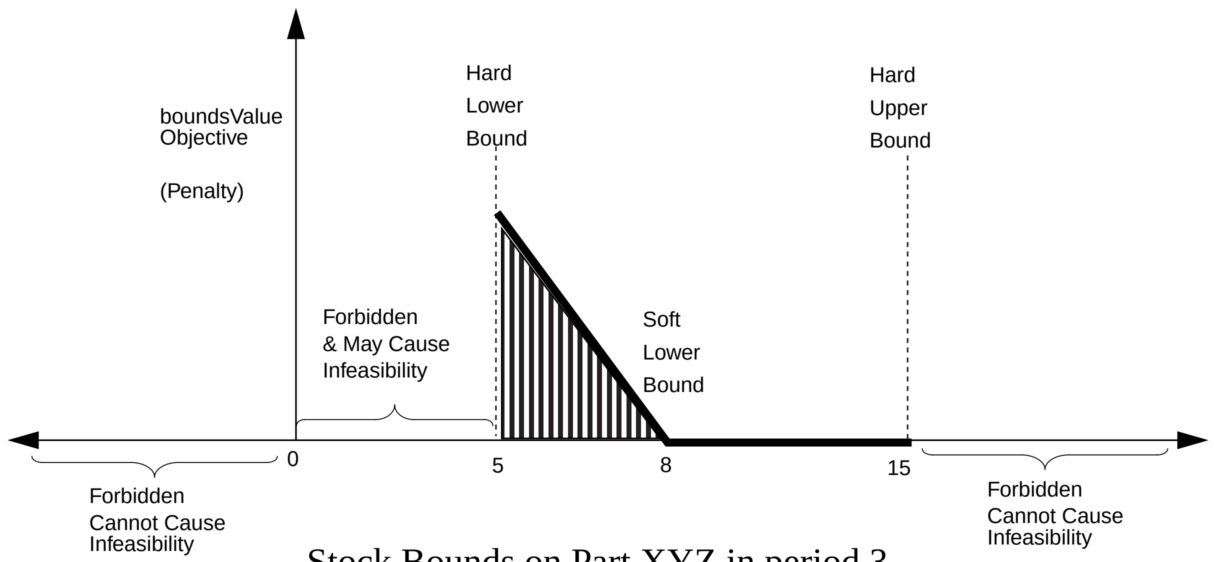
Bound Set Recommendations

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



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 weight on the bounds objective, wbounds, works a little differently from the other weights in Objective #2. While the other objectives are designed to have a “normal” weight of 1.0 (e.g., obj2Wrev = 1.0), 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 rest of Objective #2 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.

Optimizing Implosion Objective #1

Objective #1 is an advanced feature of WIT. Its purpose is to allow the user to set the cost and reward coefficients of each variable in the linear programming formulation. This section describes an essential version of Objective #1, called the primary objective, obj1PrimaryValue which assumes that the implosion problem includes no soft lower bounds. After this, the extension to Objective #1 for soft lower bounds is described.

Objective #1 without Soft Lower Bounds

The data for the primary objective is as follows:

- For each material part, `j`, and each period, `t`:
  $\texttt{obj1StockCost}_{\texttt{j,t}}$
  `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`:
  $\texttt{obj1ScrapCost}_{\texttt{j,t}}$
  `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`:
  $\texttt{obj1ShipReward}_{\texttt{i,t}}$
  `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`:
  $\texttt{obj1CumShipReward}_{\texttt{i,t}}$
  `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`:
  $\texttt{obj1ExecCost}_{\texttt{h,t}}$
  `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`:
  $\texttt{obj1SubCost}_{\texttt{s,t}}$
  `float`
  Let operation `h` be the consuming operation for substitute BOM entry `s`. Then $\text{obj1SubCost}_{\texttt{s, t}}$ 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:

`obj1PrimaryValue =`

$- \sum_{j,t} \texttt{obj1StockCost}_{j,t}$ * stock volume of part j in period t

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

$- \sum_{j,t} \texttt{obj1ScrapCost}_{j,t}$ * scrap volume of part j in period t

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

$+ \sum_{i,t} \texttt{obj1ShipReward}_{i,t}$ * shipment volume to demand stream i in period t

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

$+ \sum_{i,t} \texttt{obj1CumShipReward}_{i,t}$ * cumulative shipment volume to demand stream i in period t

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

$- \sum_{h,t} \texttt{obj1ExecCost}_{h,t}$ * execution volume of operation h in period t

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

$- \sum_{s,t} \texttt{obj1SubCost}_{s,t}$ * 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, Objective #1 is just:

objValue = obj1PrimaryValue

Recommendations for Objective #1:

- We recommend that all costs in objective #1 be specified as $\geq$ 0.0. This applies to the following:
  - obj1StockCost
  - obj1ScrapCost
  - obj1ExecCost
  - obj1SubCost

  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 obj1ExecCost < 0.0 for an operation, and the operation has an empty bill of manufacturing, and execEmptyBOM is TRUE, and obj1ScrapCost = 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 107. 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 in objective #1 are:
  - Specify obj1CumShipReward$_{i,t} > 0$ for all demands and periods, or
  - Specify obj1ShipReward$_{i,t}$ > obj1ShipReward$_{i,t+1} > 0$ for all demands and periods.

Objective #1 with Soft Lower Bounds

If the input data includes soft lower bounds (that are larger than the corresponding hard lower bounds), Objective #1 is extended to include a bounds objective, boundsValue, and a weight on the bounds objective, wbounds.

The bounds objective and wbounds are defined and used in exactly the same way in Objective #1 as in Objective #2; for more information see "Objective #2 with Soft Lower Bounds" on page 24.

With soft lower bounds, Objective #1 is defined as follows:

objValue = obj1PrimaryValue - wbounds $*$ boundsValue

Scrapping

Optimizing implosion allows for scrapping of unused quantities of a part. But the scrapping is allowed only under certain conditions, essentially only when the unneeded quantities of the part are first becoming available. The idea is to avoid keeping unneeded quantities of the part in inventory, if they would just be scrapped later. Specifically, optimizing implosion will allow the scrapping of a part in period t, if and only if any of the following conditions hold:

- The supplyVol of the part in period t is positive.
- The soft lower bound on the stockVol of the part in period t is less than in period t-1. (The difference may need to be scrapped.)
- The soft lower bound on the execution of an operation that produces the part in period t is positive.

- There is some operation that either produces or negatively consumes both the part in question and some other part. (An operation that negatively consumes a part is one that consumes it with a negative consRate.) In this case, optimizing implosion might produce the part and then scrap it, simply to enable the other part to be produced.

Heuristic implosion follows similar rules for scrapping.

This concludes the Technical Descriptions for advanced WIT users.

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
  - 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
- For Heuristic Implosion:
  - Automatic Priority
  - Equitable Allocation Heuristic Implosion
  - 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

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 7 and Table 2 show the prerequisite relationships between the different types of objects.

FIGURE 7 **Prerequisite Relationships Between Data Object Types**

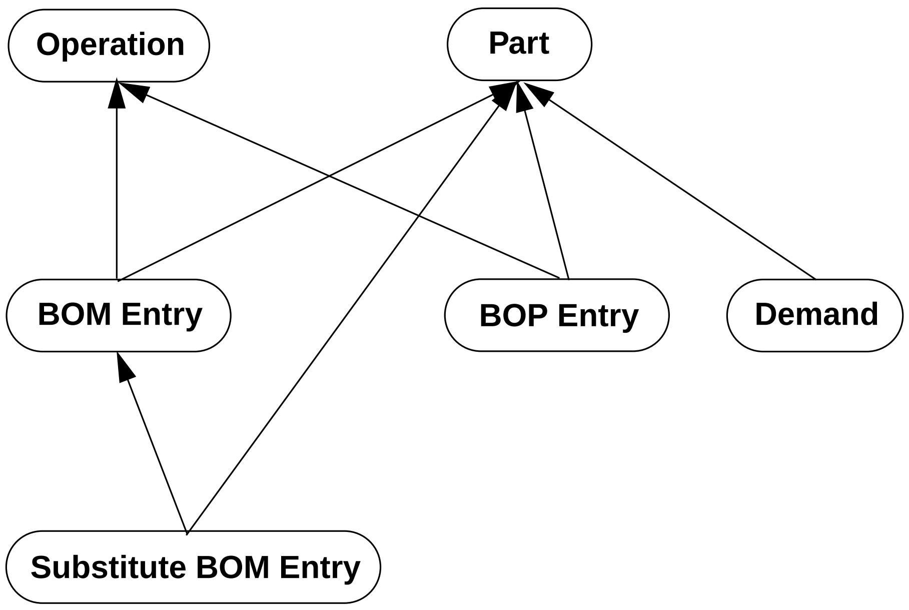


TABLE 2 **Prerequisite Relationships Between Data Object Types**

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 7 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 7 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 31.)
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 289.

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 solution. 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 36.
- The objective functions for it can be evaluated: See “Evaluating the Objective Functions of a User-Specified Solution” on page 46.
- It can be used as the starting solution for optimizing implosion: See “User-Specified Starting Solution for Optimizing Implosion” on page 47.
- It can be used as the starting solution for heuristic implosion: See “User-Specified Heuristic Starting Solution” on page 56.

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 134. 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 58.

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. However, WIT-MRP ignores substitute BOM entries. 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.

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 122.) 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 92.

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 $-1 \leq \text{focusHorizon}_i \leq \text{nPeriods} - 1$. 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 $\text{focusHorizon}_i \geq 0$ 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` $\leq \text{focusHorizon}_i$, 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 $\text{focusShortageVol}_{j,t}$ 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, $\text{focusHorizon}_{\mathtt{i}}$ = 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 134. 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 36.) 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 at most one shipment 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 individual shipments; they merely generate a feasible solution to the implosion problem (guided by the priorities, objective functions, etc.). WIT provides two alternative techniques for defining and computing this designation: “post-implosion pegging” (described in this section) and “concurrent pegging” (described in a later section).

The “post-implosion pegging” technique (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 90.) 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 is often 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`

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 80.).
- 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.

To understand how PIP works on problems with side-effects, consider problems that satisfy the following “single explodeable output” property:

- No operation has more than one BOP entry 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's 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`

Finally, consider problems that do not satisfy the "single explodeable output" property. To handle this case, WIT defines the "pipShare" attribute for BOP entries. (See "pipShare" on page 126.) This attribute is a vector of non-negative floats. If the pipShare = 0, the BOP entry considered to be a side-effect BOP entry and handled as above.

Consider an operation (and period) that has at least one BOP entry with a positive pipShare. Define the prodVol of a BOP entry to be the portion of the produced part's prodVol that's due to the BOP entry. PIP treats some of the prodVol of a BOP entry as peggable production and it treats the rest of the prodVol of a BOP entry as sideVol. The peggable production of each BOP entry is proportional to its pipShare. Let totPipShare = the sum of the pipShares over all of the BOP entries for the operation. Then the portion of the prodVol that's treated as peggable is given by:

(pipShare / totPipShare) ∗ prodVol

The portion of the prodVol that's treated as sideVol is given by:

[1 - (pipShare / totPipShare)] ∗ prodVol

Similarly to the "single explodeable output" case, an auxiliary problem can be defined for this general case, and the pegging produced by the PIP algorithm is a feasible partitioning of the original solution with respect to the auxiliary problem.

NOTES:

1. The following API functions pertain to PIP:
   - `witClearPipSeq`
   - `witAppendToPipSeq`
   - `witSetPipSeqFromHeur`
   - `witGetPipSeqFromHeur`
   - `witGetPipSeq`
   - `witSetBopEntryPipShare`
   - `witGetBopEntryPipShare`
   - `witBuildPip`
   - `witGetPipExists`
   - `witGetDemandExecVolPip`
   - `witGetDemandSubVolPip`
   - `witGetDemandSupplyVolPip`
   - `witGetDemandProdVolPip`
   - `witGetDemandConsVolPip`
   - `witGetDemandSideVolPip`
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 134.)
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 90.) 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`.

Evaluating the Objective Functions of a User-Specified Solution

The concept if a user-specified solution was introduced on page 36. Another use of a user-specified solution is to have WIT compute the value of the objective functions corresponding to it. The selection of which objectives are to be computed by this feature depends on which of the objective functions was specified in the input:

- If objective #1 was specified, then obj1PrimaryValue, boundsValue, and objValue are computed.
- If objective #2 was specified, then obj2RevValue, obj2InvValue, obj2ServValue, obj2SubValue, boundsValue, and objValue are computed.

This feature is available in API mode by calling the function, `witEvalObjectives`. See “Action Functions” on page 237.

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 36 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 87.) 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 48.

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 OSL environment are kept in memory. Then, when the second optimizing implosion is requested, the LP model and OSL 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 4 on page 135, 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 50).

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.

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 78.

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 (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 81.

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 51.) 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.
- 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 91), `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 56)

<u>Equitable Heuristic Allocation</u>

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 (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. Like heuristic implosion, heuristic allocation will try to produce “just-in-time”, in order to achieve desIncVol. If there are not enough resources available to achieve desIncVol by just-in-time production, it will try to produce in progressively earlier periods until desIncVol is achieved or buildAheadUB is reached. However, it will never build the part later than shipPeriod, because it is only allowed to ship in shipPeriod. Thus heuristic allocation ignores shipLateUB. Of course, the application program is free to request late shipments, simply by specifying a shipPeriod that’s later than the demand period. In effect, the decision of how to do late shipments constitutes the application program’s backlogging policy.
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. Although heuristic allocation (like heuristic implosion) ignores bounds on cumulative shipment, it is possible for the application program to enforce these bounds by choosing appropriate values for desIncVol.
5. 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 81.

6. 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.
7. It is permissible to intermix calls to `witIncHeurAlloc` with calls to `witEqHeurAlloc`.

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 94.) 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 36.) 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.

NOTES:

1. 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 36 for WIT’s feasibility criteria.
2. This capability can be used to achieve the effect of altering the implosion problem part-way through the heuristic allocation process, by employing the following procedure:
   - Run heuristic allocation until some change is needed in the input data.
   - Stop and modify the problem as needed (e.g., by changing supplyVols, demandVols, adding or deleting parts or BOM entries, etc.).
   - Make corresponding modifications to the solution, if necessary to keep it feasible.
   - Restart heuristic allocation using userHeurStart and continue where it left off.

   When interrupting heuristic allocation to modify the data, it may be helpful to use the `witShutDownHeurAlloc` function, instead of `witFinishHeurAlloc`. Since this avoids post-processing, the CPU time will be reduced and this may be significant, particularly if multiple interruptions are to be done.

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 42 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 89.

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 64. 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 37), 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 90.) 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 84.

<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:

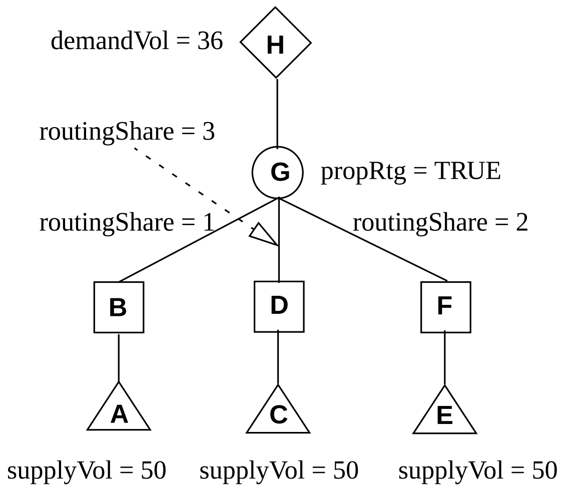
- A part produced by more than one BOP entry
- A BOM entry with at least one substitute

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 99.) 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 127.)

Consider the following example:

FIGURE 8 First Example of Proportionate Routing



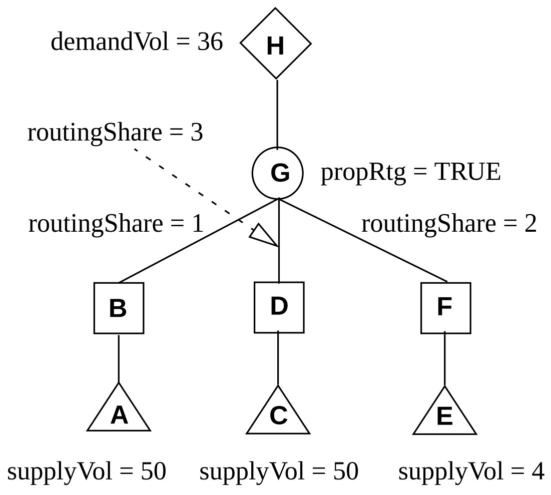
In this case, the heuristic will meet the entire demand of 36 by using BOP entries on operations B, D, and F, each in proportion to its routingShare: 1:3:2. Thus:

- 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 9 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 117.) 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 117 and on page 123.)
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 entryin 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 122. 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

<u>NSTN Build-Ahead</u>

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 97.)

<u>ASAP Build-Ahead</u>

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 97.)

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 92.

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 126.) 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 “multiExec” on page 84.) 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.

<u>Two-Way Multiple Execution Periods</u>

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 93.) 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

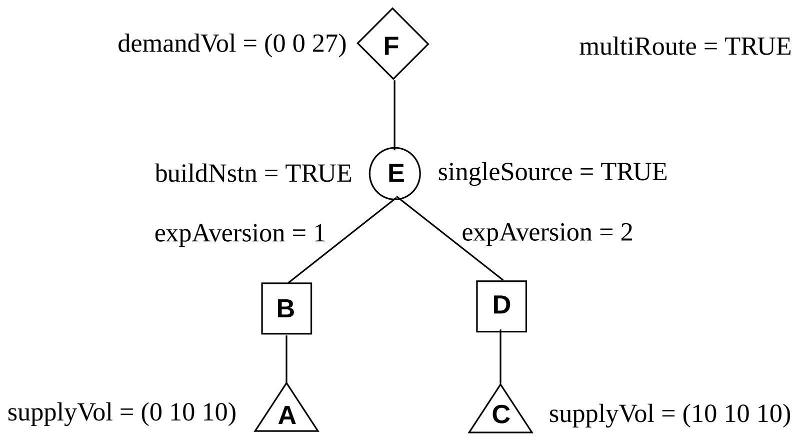
This feature only applies to heuristic implosion and heuristic allocation, and only when the multiple routes technique is being used.

The single-source technique is controlled by the singleSource boolean input attribute for parts and BOM entries. (See the singleSource attribute on page 101 and on page 118.) 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 10 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 107.) 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

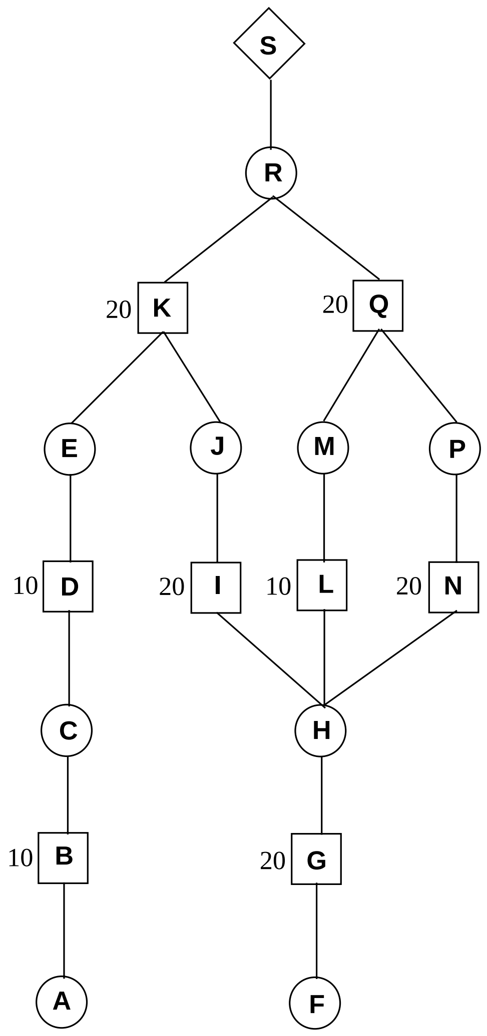
where nPaths is the number of paths in the routing by which the demand at the top of the routing uses the operation. The penalized execution technique is a heuristic for selecting a routing that has the minimum execution penalty.

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 11 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 115 and page 120.) 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 122.)

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 84 and “penExec” on page 89.

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

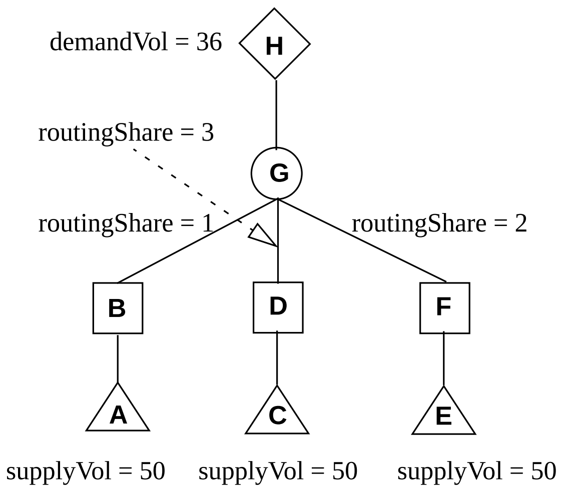
(See also “Proportionate Routing” on page 60.) The default mode is overriding proportionate routing. In this mode, proportionate routing overrides the execution penalties. Specifically, whenever the heuristic is trying to increase production of a part in a period for which the propRtg attribute is TRUE, 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 92.) 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 12 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

Object Type	Object ID	Attribute	Value
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:

Case 1:

- `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`

Case 2:

- `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`

Case 3:

- `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:

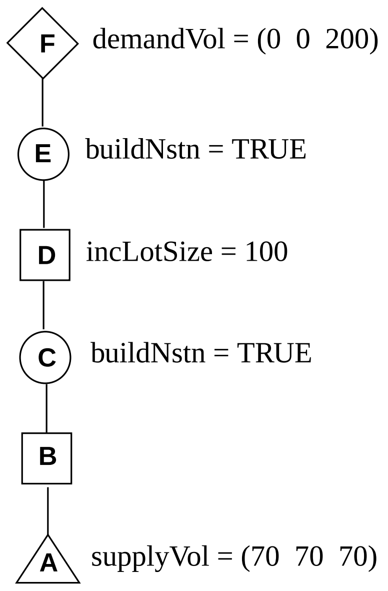
- `B.execVol = 12`
- `D.execVol =  0`
- `F.execVol = 24`

Selection Splitting

This feature only applies to heuristic implosion and allocation

Consider the following example:

FIGURE 13 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 92.) 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
2. The selection splitting technique can seriously increase the run time of heuristic implosion and allocation.

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 49.

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 50.

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 134.

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 95.

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.

capCost

Input

Float ≥ 0.0

Default value: 8.0

The annual percentage interest rate used for most purposes in objective #2. See “Optimizing Implosion Objective #2” on page 15. A value of 0.0 is allowed, but not recommended. See “Optimizing Implosion Objective #2” on page 15.

compPrices

Input

Boolean

Default value: FALSE

TRUE indicates that optimizing implosion is to compute shadow prices. See “shadowPrice” on page 100.

computeCriticalList

Input

Boolean

Default value: FALSE

TRUE indicates that the implosion functions will compute the Critical Parts List. See “Critical Parts List” on page 37.

criticalList

Output

List of part names and periods

This is the “Critical Parts List”. See “Critical Parts List” on page 37.

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 51 and "Equitable Heuristic Allocation" on page 54.

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 113 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.

The following Technical Descriptions are for advanced WIT users.

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 infeasible, because it has small negative scrapVols.)

This concludes the Technical Descriptions for advanced WIT users.

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 134. 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 51.)

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 54.)

hashTableSize

Input

Integer > 0

Default value: 2000

The initial size of the hash tables used by WIT. A reasonable size is the estimated number of parts. It’s not usually necessary to set this attribute, because WIT’s hash tables automatically expand as needed.

heurAllocActive

Output

Boolean

TRUE, if and only if heuristic allocation is active. See “Heuristic Allocation” on page 52.

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 258.

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.

invCost

Input

Float ≥ 0.0

Default value: 10.0

The annual percentage interest rate used for inventory in objective #2. This includes the time-value of the money invested in the material stored as well as the cost of storage space, etc., all expressed as an interest rate relative to the value of the material being stored in inventory. invCost should be larger than capCost. See “Optimizing Implosion Objective #2” on page 15. A value of 0.0 is allowed, but not recommended. See “Optimizing Implosion Objective #2” on page 15.

The following Technical Descriptions are for advanced WIT users.

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 108, “minLotSize” on page 110, and “incLotSize” on page 109.) 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 ≥ 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.

This concludes the Technical Descriptions for advanced WIT users.

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 64.

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 59.

nPeriods

Input

Integer > 0

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.

obj2InvValue

Output

Float

The value of the inventory objective of Objective #2. This value is meaningful only after an optimizing implosion has been performed with objChoice = 2.

obj2RevValue

Output

Float

The value of the revenue objective of Objective #2. This value is meaningful only after an optimizing implosion has been performed with objChoice = 2.

obj2ServValue

Output

0.0 ≤ Float ≤ 1.0

The value of the serviceability objective of Objective #2. This value is meaningful only after an optimizing implosion has been performed with objChoice = 2.

obj2SubValue

Output

Float ≥ 0.0

The value of the substitute objective of Objective #2. This value is meaningful only after an optimizing implosion has been performed with objChoice = 2.

obj2Winv

Input

Float ≥ 0.0

Default value: 1.0

The weight on the inventory objective of objective #2.

See “Optimizing Implosion Objective #2” on page 15. The “normal” value is 1.0. Higher values give more importance to the inventory objective; lower values give less importance to it. A value of 0.0 will cause WIT to ignore the inventory objective, but this is not recommended. See “Recommendations for Objective #2:” on page 22.

obj2Wrev

Input

Float ≥ 0.0

Default value: 1.0

The weight on the revenue objective of objective #2.

See “Optimizing Implosion Objective #2” on page 15. The “normal” value is 1.0. Higher values give more importance to the revenue objective; lower values give less importance to it. A value of 0.0 will cause WIT to ignore the revenue objective, but this is not recommend. See “Recommendations for Objective #2:” on page 22.

obj2Wserv

Input

Float ≥ 0.0

Default value: 1.0

The weight on the serviceability objective of objective #2.

See “Optimizing Implosion Objective #2” on page 15. The “normal” value is 1.0. Higher values give more importance to the serviceability objective; lower values give less importance to it. A value of 0.0 will cause WIT to ignore the serviceability objective, but this is not recommended. See “Recommendations for Objective #2:” on page 22.

obj2Wsub

Input

Float ≥ 0.0

Default value: 1.0

The weight on the substitute objective of objective #2.

See “Optimizing Implosion Objective #2” on page 15. The “normal” value is 1.0. Higher values give more importance to the substitute objective; lower values give less importance to it. A value of 0.0 will cause WIT to ignore the substitute objective (which may be appropriate in some cases).

objChoice

Input

Integer = 0, 1, 2

Default value: 2

Objective function. See “Optimizing Implosion Objective #1” on page 27 and “Optimizing Implosion Objective #2” on page 15.

0: No objective specified.

(Use only if optimizing implosion will not be used.)

1: Objective #1 specified.

2: Objective #2 specified.

If objChoice = 0, then optimizing implosion cannot be invoked and automatic priority cannot be used.

objItrState

Output

WIT-specific attribute type

This attribute identifies the current state of the object iteration process. (See “Object Iteration” on page 34.) 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.

optInitMethod

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

Value = heuristic, accelerated, schedule, or crash

Default value: heuristic

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

• heuristic	Before the optimizing implosion is performed, a heuristic implosion is performed and its solution is used as the initial solution for optimizing implosion. This is WIT’s usual method of computing 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 134.
• schedule	The execution and shipment schedules currently in WIT are used as the initial solution for the optimizing implosion.
• crash	(For advanced users.) OSL’s “crash basis” is used as the initial solution for optimizing implosion. (See the OSL manual for an explanation of “crash basis”.) This can be useful as an alternative to “heuristic optInitMethod” for problems where the heuristic implosion performs poorly.

WIT automatically sets this attribute according to the following rules:

- It is set to “heuristic” by default.
- Whenever WIT is put into an accelerated state, optInitMethod is set to “accelerated”. See “The State of a WitRun” on page 134.
- Whenever WIT is changed from an accelerated state to an unaccelerated state, if the value of optInitMethod is “accelerated”, it is set to “heuristic”.
- Whenever the execVol, subVol, or shipVol attributes are set, optInitMethod is set to “schedule”. See “User-Specified Starting Solution for Optimizing Implosion” on page 47.
- WIT never sets optInitMethod to “crash” automatically.

In API mode, these rules can be overridden, simply by setting the optInitMethod attribute. However, the optInitMethod attribute method cannot be set to “accelerated” by the user.

oslMesgFileName

Input

Character string

Default value depends on the platform.

Name of the file where WIT will cause OSL messages to be written.

The acceptable values depends on the platform.

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 360 of Appendix B and “Shipment Schedule Output File” on page 363 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 56.

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 67.

periodsPerYear

Input

Float > 0.0

Default value: 52.0

The number of periods in a year. If the period is a week, then periodsPerYear is 52. See “Optimizing Implosion Objective #2” on page 15.

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 58.

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 58.

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 42.

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 42.

postprocessed

Output

Boolean

TRUE indicates the 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 134.

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.

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 91.

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 73.

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 64.

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 70.

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 65.

✓

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 56.

wbounds

Input

Float ≥ 0.0

Default value: 10000.0

The weight on the bounds objective. See “wbounds Calculation and Recommendation” on page 26.

wit34Compatible

Input

Boolean

Default value: FALSE

The purpose of this attribute is to assist former users of WIT 3.4, the IBM internal version of WIT that preceded WIT release 4. TRUE indicates that WIT will operate in WIT 3.4 compatible mode. This means that it will be allowable to invoke API functions that existed in WIT 3.4, but do not normally apply to WIT 5.0. In this case, the 3.4-only functions simply call the corresponding 5.0 functions. If wit34Compatible is FALSE, invoking these functions generates a severe error.

Part Attributes

partName

Immutable input attribute

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.

The following Technical Descriptions are for advanced WIT users.

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.

This concludes the Technical Descriptions for advanced WIT users.

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.

buildAheadUB

Input

Vector of integers with length equal to nPeriods

$0 \leq$ buildAheadUB$_t \leq$ nPeriods - 1   For all t = 0, 1, … nPeriods - 1

Default value: nPeriods - 1

Defines an upper bound on the number of periods by which heuristic build-ahead can be done for the part. (See “Build-Ahead” on page 63.) When heuristic implosion or allocation is doing 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. 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:

buildAheadUB$_t \leq$ buildAheadUB$_{t-1}$ + 1   For all t = 1, 2, … nPeriods - 1

This condition simply disallows build-ahead for period t in any period earlier than is allowed for period t-1.

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 63.

Defined only for parts with partCategory = material. (Build-ahead does not make sense for capacity parts.)

See also the note on page 97.

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 63.

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 134.

excessVol

Output

Vector of floats with length equal to nPeriods

$\text{excessVol}_t \geq 0.0$ 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 100. This attribute is only valid when WIT is in a postprocessed state. See “The State of a WitRun” on page 134.

excessVol is related to residualVol as follows:

$\text{excessVol}_t = \min(\text{residualVol}_t, \text{supplyVol}_t)$

focusShortageVol

Output

Vector of floats with length equal to nPeriods

$\text{focusShortageVol}_t \geq 0.0$ 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 91.

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

$\text{mrpExcessVol}_t \geq 0.0$ 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 98. 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 98. This attribute is valid when WIT-MRP has been invoked.

obj1ScrapCost

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 objective #1. Negative values are allowed, but not recommended; see "Recommendations for Objective #1:" on page 28

obj1StockCost

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 objective #1. Negative values are allowed, but not recommended; see "Recommendations for Objective #1:" on page 28

Defined only for parts with partCategory = material

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 60.

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 91.

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 97. This attribute is only valid when WIT is in a postprocessed state. See “The State of a WitRun” on page 134.

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 134.

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 31.

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 79.

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 66.

stockBounds

Input

Bound set

Default value: See “Bound Set Attributes” on page 128.

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

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 134.

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.

unitCost

Input

Float ≥ 0.0

Default value: 1.0

The standard unit cost of the part. See “Optimizing Implosion Objective #2” on page 15.

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

Immutable input attribute

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 95.

cumShipBounds

Input

Bound set

Default value: See “Bound Set Attributes” on page 128.

Bounds on the cumulative shipment of part to the demand.

demandVol

Input

$\text{demandVol}_t \geq 0.0$ 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

$\text{fssShipVol}_t \geq 0.0$ 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.

grossRev

Input

Float $\geq$ 0

Default value: 1.1

The gross revenue per unit of the part shipped to the demand. If this is less than the standard unit cost, WIT acts as if the gross revenue was equal to the standard unit cost. This means the net revenue (grossRev - unitCost) is never negative. See “Optimizing Implosion Objective #2” on page 15.

obj1CumShipReward

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, $\text{obj1CumShipReward}_t$ is the per-unit reward for the total volume shipped to the demand stream during periods 0,1,...,t. This is used by objective #1.

See also “Recommendations for Objective #1:” on page 28.

obj1ShipReward

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, obj1ShipReward_t is the reward received for each unit shipped to the demand stream in period t. This is used by objective #1.

See also “Recommendations for Objective #1:” on page 28.

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.

selForDel

Same definition as in “Part Attributes” on page 95.

shipLateUB

Input

Vector of integers with length equal to nPeriods

$0 \leq \text{shipLateUB}_t \leq \text{nPeriods} - 1$ For all t = 0, 1, … nPeriods - 1

Default value: nPeriods - 1

Defines an upper bound on the number of periods by which 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:

$\text{shipLateUB}_t \geq \text{shipLateUB}_{t-1} - 1$ 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.

shipVol

Normally an output, but may be an input. See “User-Specified Solution” on page 36.

Vector of floats with length equal to nPeriods.

$\text{shipVol}_t \geq 0.0$

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

Immutable input attribute

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 95.

execBounds

Input

Bound set

Default value: See “Bound Set Attributes” on page 128.

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.

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 67.

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:

- $\text{yieldRate}_t = 0.0$
- 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 36.

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.

lotSize2Thresh

Input

Vector of floats with length equal to nPeriods

$\text{lotSize2Thresh}_t \geq 0.0$ For all t = 0, 1, … nPeriods - 1

Default value: Vector of all 0.0’s

Applies during heuristic implosion and WIT-MRP. 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 $\text{execVol}_t \geq \text{lotSize2Thresh}_t$, then minLotSize2_t and incLotSize2_t are applied to execVol_t.

If twoLevelLotSizes is TRUE and $\text{execVol}_t < \text{lotSize2Thresh}_t$, then minLotSize_t and incLotSize_t are applied to execVol_t.

minLotSize

Input

Vector of floats with length equal to nPeriods

$\text{minLotSize}_t \geq 0.0$ 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 109.

minLotSize2

Input

Vector of floats with length equal to nPeriods

$\text{minLotSize2}_t \geq 0.0$ 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 109.

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.

obj1ExecCost

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 objective #1. Negative values are allowed, but not recommended; see "Recommendations for Objective #1:" on page 28

obj2AuxCost

Input

Float ≥ 0.0

Default value: 0.0

The "auxiliary cost" of the operation. See "Optimizing Implosion Objective #2" on page 15.

selForDel

Same definition as in "Part Attributes" on page 95.

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 110.

If twoLevelLotSizes is TRUE, the following restrictions are enforced in all periods, t:

$incLotSize_t \geq 1.0$

$incLotSize2_t \geq 1.0$

yieldRate

Input

Vector of floats with length equal to nPeriods

0.0 1≤ $yieldRate_t \leq 1.0$ or $yieldRate_t = 0.0$ 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 127.

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 124.

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 95.

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:

$execVol_t * consRate_t / (1 - falloutRate)$

earliestPeriod

Input

$0 \leq$ Integer $\leq$ 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.

☞

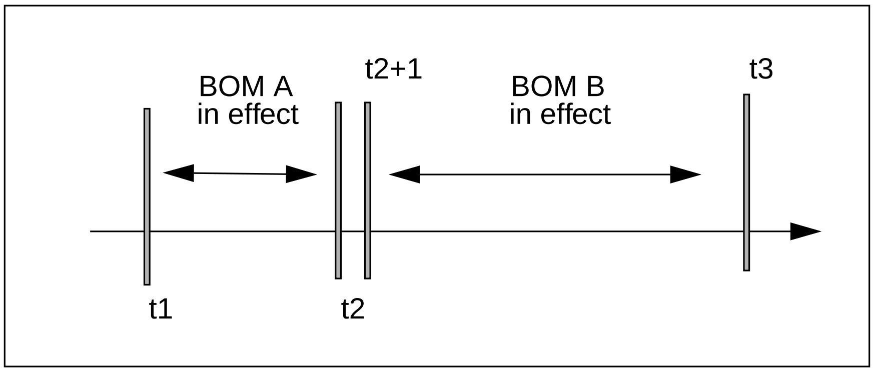
Consider the following example:

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 14 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 67.

falloutRate

Input

Float

$0.0 \leq \text{falloutRate} < 0.99$

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 114.

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 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. Otherwise, the value is rounded to an integer, according to WIT's rounding procedure. See "offset" on page 116.

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 114.

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 108.

Offsets are specified as floating point numbers rather than integers. This allows for the possibility that the offset data is obtained from another database that uses smaller time units than the “period” used by the WIT. For example, if the period for WIT purposes is a 5-day week, and data from your database specifies offsets in days, then divide by 5. If the result of dividing is not an integer, one can proceed in either of two ways:

1. One approach is to specify the non-integer offset to WIT. WIT will take the non-integer offset and round it either up or down to an integer. The choice of rounding up vs. down will be done by a heuristic whose purpose is to avoid cumulative round-off errors as one proceeds down the multi-level BOM.
2. Alternatively you can round the fractional offset yourself and specify the integer result to WIT (as a float whose value coincides with an integer). In this offset case, you are taking responsibility for doing the rounding in a way that avoids cumulative round-off errors.

See also “impactPeriod” on page 115.

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 60.

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 60.

selForDel

Same definition as in “Part Attributes” on page 95.

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 66.

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 113.
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 124.

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 95.

consRate

Same definition as in “BOM Entry Attributes” on page 113.

earliestPeriod

Same definition as in “BOM Entry Attributes” on page 113.

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 67.

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 59.) 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 113.

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 113.

latestPeriod

Same definition as in “BOM Entry Attributes” on page 113.

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

$mrpSubVol_t \geq 0.0$ 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.

obj1SubCost

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 $obj1SubCost_t$ 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 Objective #1:” on page 28

obj2SubPenalty

Input

Float ≥ 0.0

Default value: 0.0

The penalty associated with using the substitute BOM entry. See “Objective #2 with Substitutes” on page 23. This is used by objective #2.

offset

Input or Output, depending of the value of the global independentOffsets attribute. See also “independentOffsets” on page 82. 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 108.

Offsets are specified as floating point numbers rather than integers. This allows for the possibility that the offset data is obtained from another database that uses smaller time units than the “period” used by the WIT.

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 60.

selForDel

Same definition as in “Part Attributes” on page 95.

subVol

Normally, an output, but may be an input. See “User-Specified Solution” on page 36.

Vector of floats with length equal to nPeriods.

$\text{subVol}_t \geq 0.0$ 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 113.
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 125. 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.

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 95.

earliestPeriod

Same definition as in “BOM Entry Attributes” on page 113.

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 124.

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.

$-1 \le \text{impactPeriod}_t < \text{nPeriods}$

Given any period t, where $0 \le t < \text{nPeriods}$, 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 $\text{impactPeriod}_t \ge 0$, 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 115.

latestPeriod

Same definition as in “BOM Entry Attributes” on page 113.

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 108.

Offsets are specified as floating point numbers rather than integers. This allows for the possibility that the offset data is obtained from another database that uses smaller time units than the “period” used by the WIT. See also “offset” on page 116.. See also “impactPeriod” on page 126.

pipShare

Input

Vector of floats with length equal to nPeriods

$\text{pipShare}_t \ge 0.0$ For all t = 0, 1, … nPeriods - 1

Default value: Vector of all 1.0’s

Applies to post-implosion pegging. For any execution period, t, the portion of the production from this BOP entry that’s treated as peggable is proportional to

$pipShare_t$, while the rest of the production is treated as a side-effect. See "Post-Implosion Pegging" on page 42.

productRate

Input

Vector of floats with length equal to nPeriods

$prodRate_t \geq 0.0$ 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:

$execVol_t * productRate_t * yieldRate_t$

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 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 60.

selForDel

Same definition as in "Part Attributes" on page 95.

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 3 on page 130. 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 354 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 3 Input and Output File Descriptions (Stand-Alone Executable)

File	Direction	Control Parameter	Content
Control Parameter File	Input	none, see page 129.	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.
OSL Log File	Output	osl_ofname	Messages from OSL. (This file is produced only if optimizing implosion is used.)
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`. 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`.

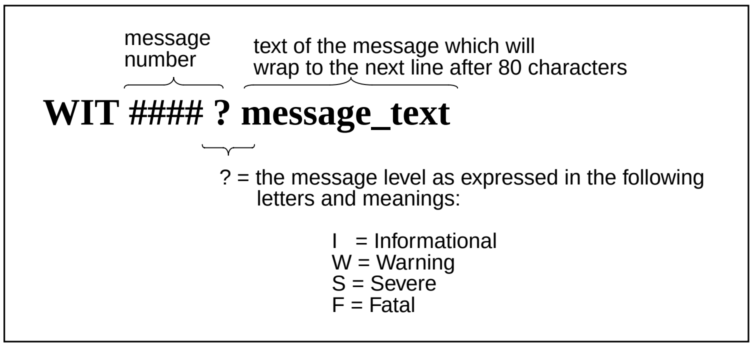
  Since the relation

```
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.

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 prevents WIT from continuing. The default action has WIT terminating the run 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 running the application. Under these circumstances the application is terminated when `witPreprocess` is invoked. This allows the application to identify all out-of-range values with a single run.

  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`.
- **F** is fatal. Fatal messages indicate that WIT has recognized a condition which probably represents an internal programming error.

mesgFileAccessMode

`char *`

Default value: "a"

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. The value `WitSTDOUT` 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`, (which is the default for these messages), WIT will cause the application program to stop running after issuing the message. If the application program is to regain control after a severe or fatal message is 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).

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.

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 128.

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 4 on page 135.

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 38. 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 78.) and their values can be obtained by the appropriate API “get” routine. (See “witGetAttribute” on page 141.) 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 4 on page 135. 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 4 **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?
accAfterSoftLB	Global	No
accAfterOptImp	Global	Setting it to True preserves an accelerated state. Setting it to False puts the WitRun in an unaccelerated state.

TABLE 4 **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
autoPriority	Global	Yes
buildAheadUB	Part	Yes
buildAsap	Part	Yes
buildNstn	Part	Yes
capCost	Global	Yes
compPrices	Global	Yes
computeCriticalList	Global	Yes
consRate	BOM Entry Substitute BOM Entry	No
cumShipBounds	Demand	See “Bound Sets and Accelerated Optimizing Implosion” on page 50.
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 50
execEmptyBom	Global	No
execPenalty	Operation BOM Entry Substitute BOM Entry	Yes
execVol	Operation	Yes
expAllowed	Substitute BOM Entry	Yes
expAllowed	BOP Entry	No
expAversion	BOP Entry	No
expCutoff	Global	No
expNetAversion	Substitute BOM Entry	Yes
falloutRate	BOM Entry Substitute BOM Entry	No
focusHorizon	Demand	Yes
forcedMultiEq	Global	Yes
grossRev	Demand	Yes
hashTableSize	Global	Yes
highPrecisionWD	Global	Yes
incLotSize	Operation	No
incLotSize2	Operation	No
independentOffsets	Global	No

TABLE 4 **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?
invCost	Global	Yes
latestPeriodt	BOM Entry Substitute BOM Entry BOP Entry	No
lotSize2Thresh	Operation	No
lotSizeTol	Global	Yes
mandEC	BOM Entry Substitute BOM Entry	No
minLotSize	Operation	No
minLotSize2	Operation	No
mrpNetAllowed	Substitute BOM Entry	Yes
multiExec	Global	No
multiRoute	Global	No
netAllowed	Substitute BOM Entry	Yes
nPeriods	Global	No
objChoice	Global	No
obj1CumShipReward	Demand	Yes
obj1ExecCost	Operation	Yes
obj1ScrapCost	Part	Yes
obj1ShipReward	Demand	Yes
obj1StockCost	Part	Yes
obj1SubCost	Substitute BOM Entry	Yes
obj2AuxCost	Operation	Yes
obj2SubPenalty	Substitute BOM Entry	Yes
obj2Winv	Global	Yes
obj2Wrev	Global	Yes
obj2Wserv	Global	Yes
obj2Wsub	Global	Yes
offset	BOM Entry BOP Entry Substitute BOM Entry	No
optInitMethod	Global	No
oslMesgFileName	Global	Yes
perfPegging	Global	Yes
penExec	Global	Yes
periodsPerYear	Global	Yes
pgdCritListMode	Global	Yes
pipSeqFromHeur	Global	Yes

TABLE 4 **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?
pipShare	BOP Entry	Yes
prefHighStockSLBs	Global	Yes
priority	Demand	Yes
productRate	BOP Entry	No
propRtg	Part BOM Entry	Yes
respectStockSLBs	Global	Yes
roundReqVols	Global	Yes
routingShare	BOM Entry Substitute BOM Entry BOP Entry	Yes
selForDel	Any	Yes
selSplit	Global	Yes
shipLateUB	Demand	Yes
shipVol	Demand	Yes
singleSource	Part	Yes
singleSource	BOM Entry	Yes
skipFailures	Global	Yes
stockBounds	Part	See “Bound Sets and Accelerated Optimizing Implosion” on page 50
stockReallocation	Global	Yes
subVol	Substitute BOM Entry	Yes
supplyVol	Part	Yes
tieBreakPropRt	Global	Yes
title	Global	Yes
truncOffsets	Global	No
twoLevelLotSizes	Operation	No
twoWayMultiExec	Global	No
unitCost	Part	Yes
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
- Perform object iteration
- Work with post-implosion pegging
- Control messages
- Access and modify data using double precision.

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.

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 78.

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 witGetCapCost
(    WitRun * const theWitRun,
     float * capCost );
witReturnCode witGetCompPrices
(    WitRun * const theWitRun,
     witBoolean * compPrices );
witReturnCode witGetComputeCriticalList
(    WitRun * const theWitRun,
     witBoolean * computeCriticalList );
witReturnCode witGetEquitability
(    WitRun * const theWitRun,
     int * equitability );
witReturnCode witGetExecEmptyBom
```

```
(    WitRun * const theWitRun,
     int * execEmptyBom );
witReturnCode witGetExpCutoff
(    WitRun * const theWitRun,
     float * expCutoff );
witReturnCode witGetFeasible
(    WitRun * const theWitRun,
     witBoolean * feasible );
witReturnCode witGetForcedMultiEq
(    WitRun * const theWitRun,
     witBoolean * forcedMultiEq );
witReturnCode witGetHashTableSize
(    WitRun * const theWitRun,
     int * hashTableSize );
witReturnCode witGetHeurAllocActive
(    WitRun * const theWitRun,
     witBoolean * heurAllocActive );
witReturnCode witGetIndependentOffsets
(    WitRun * const theWitRun,
     witBoolean * independentOffsets );
witReturnCode witGetInvCapCost
(    WitRun * const theWitRun,
     float * invCost );
witReturnCode witGetLotSizeTol
(    WitRun * const theWitRun,
     float * lotSizeTol );
witReturnCode witGetMultiExec
(    WitRun * const theWitRun,
     witBoolean * multiExec );
witReturnCode witGetMultiRoute
(    WitRun * const theWitRun,
     witBoolean * multiRoute );
witReturnCode witGetNPeriods
(    WitRun * const theWitRun,
     int * nPeriods );
witReturnCode witGetObj2Winv
(    WitRun * const theWitRun,
     float * obj2Winv );
witReturnCode witGetObj2Wrev
(    WitRun * const theWitRun,
     float * obj2Wrev );
witReturnCode witGetObj2Wserv
(    WitRun * const theWitRun,
     float * obj2Wserv );
```

```
witReturnCode witGetObj2Wsub
(    WitRun * const theWitRun,
     float * obj2Wsub );
witReturnCode witGetObjChoice
(    WitRun * const theWitRun,
     int * objChoice );
witReturnCode witGetOptInitMethod
(    WitRun * const theWitRun,
     witAttr * optInitMethod );
witReturnCode witGetOslMesgFileName
(    WitRun * const theWitRun,
     char * * oslMesgFileName );
witReturnCode witGetOutputPrecision
(    WitRun * const theWitRun,
     int * outputPrecision );
witReturnCode witGetPenExec
(    WitRun * const theWitRun,
     witBoolean * penExec );
witReturnCode witGetPerfPegging
(    WitRun * const theWitRun,
     witBoolean * perfPegging );
witReturnCode witGetPeriodsPerYear
(    WitRun * const theWitRun,
     float * periodsPerYear );
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 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 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 );
witReturnCode witGetWit34Compatible
(    WitRun * const theWitRun,
     witBoolean * wit34Compatible );
```

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. Concerning `witGetOslMesgFileName` and `witGetTitle` — It is the responsibility of the application to free the returned vector.

Example

```
float   capCost, wbounds;
int     nPeriods;
witBoolean accAfterOptImp;
witGetCapCost( theWitRun, &capCost );
witGetWbounds( theWitRun, &wbounds );
witGetNperiods( theWitRun, &nPeriods );
printf( "capCost = %f\n", capCost );
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 38.

`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 134.
- The current implosion solution must not be user-specified. See “User-Specified Solution” on page 36

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);
```

witGetObjValues

```
witReturnCode witGetObjValues
(    WitRun * const theWitRun,
     float * objValue,
     float * boundsValue );
```

Description

This function returns the value of the objective function value attributes, objValue and boundsValue, as computed by `witOptImplode` or `witEvalObjectives`.

`theWitRun`

Identifies the WIT problem to be used by this function.

`objValue`

On return this contains the value of the “objValue” objective function value attribute.

`boundsValue`

On return this contains the value of the “boundsValue” objective function value attribute.

Usage Notes

1. See “What Optimizing Implosion Does” on page 14 for more information on objective function values.
2. Must follow `witOptImplode` or `witEvalObjectives` or else 0.0 is returned for the objective function values.

Example

```
float objValue, boundsValue;
witGetObjValues(theWitRun, &objValue, &boundsValue );
printf( " Obj Value = %f\n", objValue );
printf( " Bounds Obj Value = %f\n", boundsValue );
```

The objective function values are obtained and printed.

witGetObj2Values

```
witReturnCode witGetObj2Values
(    WitRun * const theWitRun,
     float * obj2RevValue,
     float * obj2InvValue,
     float * obj2ServValue,
     float * obj2SubValue );
```

Description

This function returns the value of the objective function value attributes for objective #2, as computed by `witOptImplode` or `witEvalObjectives`, when `objChoice` is 2.

`theWitRun`

Identifies the WIT problem to be used by this function.

`obj2RevValue`

On return this contains the value of the “obj2RevValue” objective function value attribute.

`obj2InvValue`

On return this contains the value of the “obj2InvValue” objective function value attribute.

`obj2ServValue`

On return this contains the value of the “obj2ServValue” objective function value attribute.

`obj2SubValue`

On return this contains the value of the “obj2SubValue” objective function value attribute.

Usage Notes

1. See “Optimizing Implosion Objective #2” on page 15 for more information on Objective #2.
2. If the `objChoice` is not 2, then `obj2RevValue`, `obj2InvValue`, `obj2ServValue`, and `obj2SubValue` are returned with a value of 0.0.
3. Must follow `witOptImplode` or `witEvalObjectives` or else 0.0 is returned for the objective function values.

Example

```
float rev, inv, serv, sub;
witGetObj2Values(theWitRun, &rev, &inv, &serv, &sub );
printf ("Obj function components are: %f, %f, %f, %f\n",
          rev, inv, serv, sub );
```

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 retrives the pegged critical list. See “Pegged Criticial List” on page 58.

`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 78.

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 witSetCapCost
(    WitRun * const theWitRun,
     const float capCost );
witReturnCode witSetCompPrices
(    WitRun * const theWitRun,
     const witBoolean compPrices );
witReturnCode witSetComputeCriticalList
(    WitRun * const theWitRun,
     const witBoolean computeCriticalList );
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 witSetForcedMultiEq
(    WitRun * const theWitRun,
     const witBoolean forcedMultiEq );
witReturnCode witSetHashTableSize
```

```
(    WitRun * const theWitRun,
     const int hashTableSize );
witReturnCode witSetIndependentOffsets
(    WitRun * const theWitRun,
     const witBoolean independentOffsets );
witReturnCode witSetInvCost
(    WitRun * const theWitRun,
     const float invCost );
witReturnCode witSetLotSizeTol
(    WitRun * const theWitRun,
     const float lotSizeTol );
witReturnCode witSetMultiExec
(    WitRun * const theWitRun,
     const witBoolean multiExec );
witReturnCode witSetMultiRoute
(    WitRun * const theWitRun,
     const witBoolean multiRoute );
witReturnCode witSetNPeriods
(    WitRun * const theWitRun,
     const int nPeriods );
witReturnCode witSetObjChoice
(    WitRun * const theWitRun,
     const int objChoice );
witReturnCode witSetObj2Winv
(    WitRun * const theWitRun,
     const float obj2Winv );
witReturnCode witSetObj2Wrev
(    WitRun * const theWitRun,
     const float obj2Wrev );
witReturnCode witSetObj2Wserv
(    WitRun * const theWitRun,
     const float obj2Wserv );
witReturnCode witSetObj2Wsub
(    WitRun * const theWitRun,
     const float obj2Wsub );
witReturnCode witSetOptInitMethod
(    WitRun * const theWitRun,
     const witAttr optInitMethod );
witReturnCode witSetOslMesgFileName
(    WitRun  * const theWitRun,
     const char  * const oslMesgFileName );
witReturnCode witSetOutputPrecision
(    WitRun * const theWitRun,
     int outputPrecision );
```

```
witReturnCode witSetPenExec
(    WitRun * const theWitRun,
     const witBoolean penExec );
witReturnCode witSetPerfPegging
(    WitRun * const theWitRun,
     const witBoolean perfPegging );
witReturnCode witSetPeriodsPerYear
(    WitRun * const theWitRun,
     const float periodsPerYear );
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 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 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 );
witReturnCode witSetWit34Compatible
(    WitRun * const theWitRun,
     const witBoolean wit34Compatible );
```

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 134.
2. Concerning `witSetTitle`—A copy of the title string is made by `witSetTitle`.
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 78.

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 = witSetHashTableSize(theWitRun,5000);
rc = witSetPeriodsPerYear(theWitRun,12.0);
rc = witSetTitle(theWitRun,"Plant 3, Spring 93");
```

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 95.

Error conditions

- Any violations of the requirements listed in "Part Attributes" on page 95.
- 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 95, “Operation Attributes” on page 107, and “BOP Entry Attributes” on page 124.

Error conditions

- Any violations of the requirements listed in “Part Attributes” on page 95 and “Operation Attributes” on page 107.
- 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 95.

The `witGetPartAttribute` functions are:

```
witReturnCode witGetPartAppData
(    WitRun * const theWitRun,
     const char * const partName,
     void ** appData );
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 witGetPartObj1ScrapCost
(    WitRun * const theWitRun,
     const char * const partName,
     float ** obj1ScrapCost );
witReturnCode witGetPartObj1StockCost
(    WitRun * const theWitRun,
     const char * const partName,
     float ** obj1StockCost );
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 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 witGetPartStockVol
(    WitRun * const theWitRun,
     const char * const partName,
     float ** stockVol );
witReturnCode witGetPartSupplyVol
(    WitRun * const theWitRun,
     const char * const partName,
     float ** supplyVol );
witReturnCode witGetPartUnitCost
(    WitRun * const theWitRun,
     const char * const partName,
     float * unitCost );
```

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 134.

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 95.
- Concerning `witGetPartFocusShortageVol`— The WitRun must be in a postprocessed state. See “The State of a WitRun” on page 134.
- Also concerning `witGetPartFocusShortageVol`— The current implosion solution must not be user-specified. See “User-Specified Solution” on page 36
- Example

```
float * prodVol;
float * stockVol;
float unitCost;
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" );
witGetPartUnitCost ( theWitRun, "prod1", &unitCost );
printf ( "Unit Cost of prod1 is %f\n", unitCost );
```

The per period production and stock volume for "prod1" is obtained and printed. The part category of "partAbc" is obtained and printed. The unitCost of "prod1" 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 204 for the performance implications of this situation.

Error Conditions

- `partName` does not identify and 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`

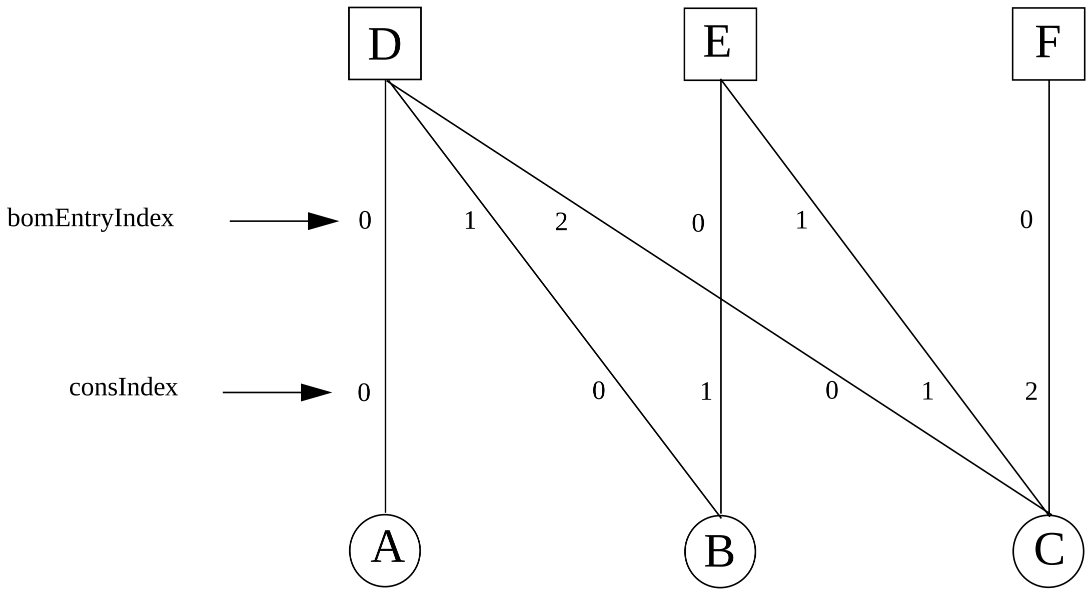
On return contains the bomEntryIndex of the specified consuming BOM entry.

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 179.

FIGURE 15 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`

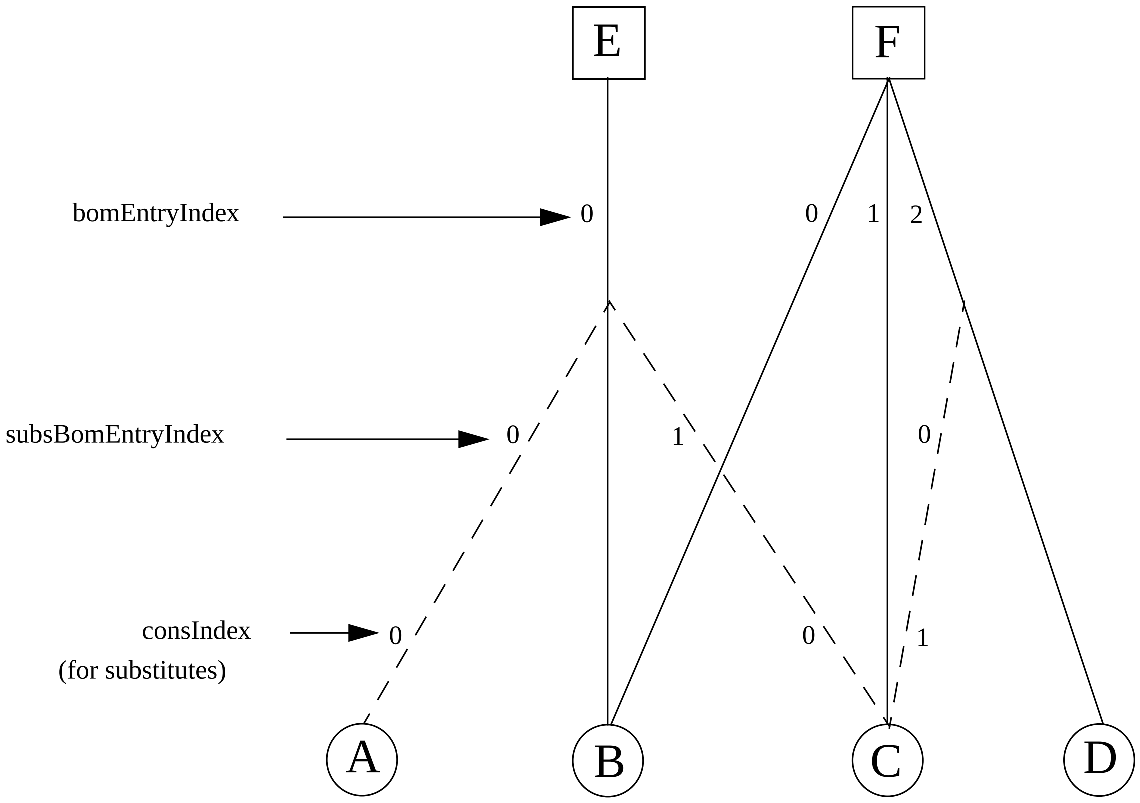
On return contains the subsBomEntryIndex of the specified consuming substitute BOM entry.

Error Conditions

- `partName` does not match the partName of an existing part.
- A `consIndex` outside the range:

  0 ≤ `consIndex` < number of consuming substitute BOM entries for the specified part. See “witGetSubsBomEntryAttribute” on page 221.

FIGURE 16 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 and 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 171.

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 174.

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 183.

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`

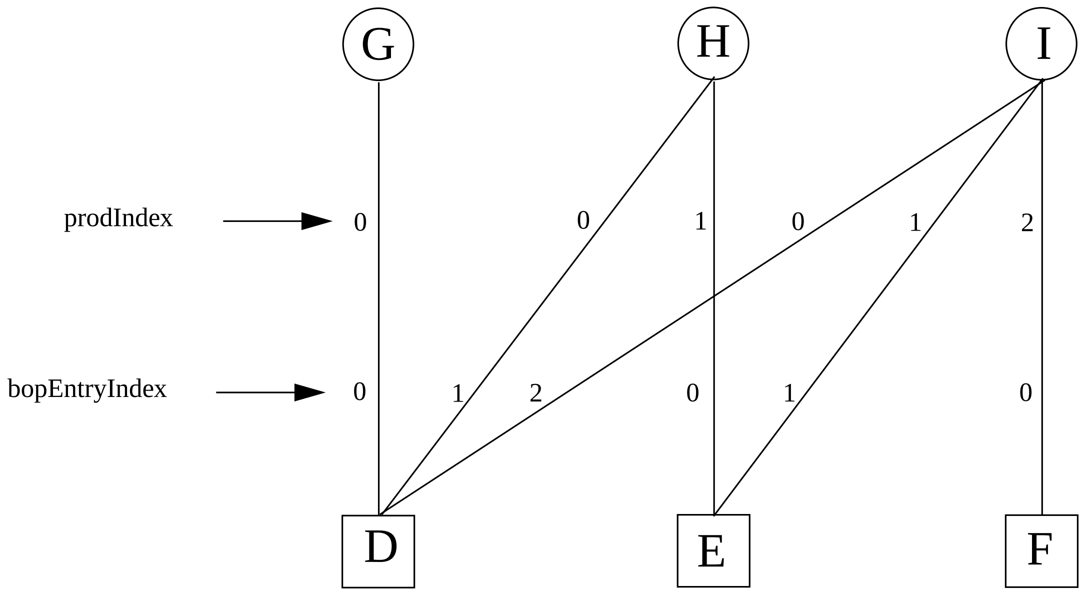
On return contains the bopEntryIndex of the specified producing BOP entry.

Error Conditions

- `partName` does not match the partName of an existing part.
- A `prodIndex` outside the range:

  0 ≤ `prodIndex` < number of producing BOP entries for the specified part. See “witGetPartNProducingBopEntries” on page 181.

FIGURE 17 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 95.

The `witSetPartAttribute` functions are:

```
witReturnCode witSetPartAppData
(    WitRun * const theWitRun,
     const char * const partName,
     void * const appData );
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 witSetPartObj1ScrapCost
(    WitRun * const theWitRun,
     const char * const partName,
     const float * const obj1ScrapCost );
witReturnCode witSetPartObj1StockCost
(    WitRun * const theWitRun,
     const char * const partName,
     const float * const obj1StockCost );
witReturnCode witSetPartPropRtg
(    WitRun * const theWitRun,
     const char * const partName,
     const witBoolean * propRtg );
witReturnCode witSetPartStockBounds
(    WitRun * const theWitRun,
     const char * const partName,
     const float * const hardLower,
     const float * const softLower,
     const float * const hardUpper );
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 );
witReturnCode witSetPartUnitCost
(    WitRun * const theWitRun,
     const char * const partName,
     const float unitCost );
```

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 134.

Error conditions

- A part with the specified `partName` has not been previously defined.
- Any violations of the requirements listed in “Part Attributes” on page 95.
- A bound set vector value which does not satisfy the requirements for a bound set. See “API Bound Set Definition” on page 134.

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

```
witReturnCode rc;
float fltv1[]    = {  1.,   3.,   5.  };
int   intv1[]    = {  7,    8,    9   };
float infinity[] = { -1.0, -1.0, -1.0 };

rc = witSetPartSupplyVol    ( theWitRun,"prod1", fltv1)
rc = 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 time periods.

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 103.

Error conditions

- Any violations of the requirements listed in “Demand Attributes” on page 103.
- 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 103.

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 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 witGetDemandGrossRev
(    WitRun * const theWitRun,
     const char  * const demandedPartName,
     const char  * const demandName,
     float * grossRev );
 witReturnCode witGetDemandObj1CumShipReward
(    WitRun * const theWitRun,
     const char  * const demandedPartName,
     const char  * const demandName,
     float * * obj1CumShipReward );
witReturnCode witGetDemandObj1ShipReward
(    WitRun * const theWitRun,
     const char  * const demandedPartName,
     const char  * const demandName,
     float ** obj1ShipReward );
witReturnCode witGetDemandPriority
(    WitRun * const theWitRun,
     const char  * const demandedPartName,
     const char  * const demandName,
     int * * priority );
witReturnCode witGetDemandSelForDel
(    WitRun * const theWitRun,
     const char  * const demandedPartName,
     const char  * const demandName,
     witBoolean * selForDel );
witReturnCode witGetDemandShipLateUB
(    WitRun * const theWitRun,
     const char  * const demandedPartName,
     const char  * const demandName,
     int ** shipLateUB );
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;
intnPeriods, 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 56. 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 191.
- `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
```

witGetDemandSubVolPegging

```
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 56. 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 191.
- `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 192 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 103.

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 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 witSetDemandGrossRev
(    WitRun * const theWitRun,
     const char * const demandedPartName,
     const char * const demandName,
     const float grossRev );
witReturnCode witSetDemandObj1CumShipReward
(    WitRun * const theWitRun,
     const char * const demandedPartName,
     const char * const demandName,
     const float * const obj1CumShipReward );
witReturnCode witSetDemandObj1ShipReward
(    WitRun * const theWitRun,
     const char * const demandedPartName,
     const char * const demandName,
     const float * const obj1ShipReward );
witReturnCode witSetDemandPriority
(    WitRun * const theWitRun,
     const char * const demandedPartName,
     const char * const demandName,
     const int * const priority );
witReturnCode witSetDemandSelForDel
(    WitRun * const theWitRun,
     const char * const demandedPartName,
     const char * const demandName,
     const witBoolean selForDel );
witReturnCode witSetDemandShipLateUB
(    WitRun * const theWitRun,
     const char * const demandedPartName,
     const char * const demandName,
     const int * shipLateUB );
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 134.
2. The attribute `fssShipVol` should normally be set after doing an implosion, and before obtaining the `FocusShortageVol`.

Error conditions

- A demand with the specified demandedPartName and demandName must have been previously defined. See also “Error Conditions” on page 191.
- Any violations of the requirements listed in “Demand Attributes” on page 103.
- A bound set vector value which does not satisfy the requirements for a bound set. See “API Bound Set Definition” on page 134.

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 107.

Error conditions

- Any violations of the requirements listed in “Operation Attributes” on page 107.
- 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 107.

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 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 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 witGetOperationObj1ExecCost
(    WitRun * const theWitRun,
     const char * const operationName,
     float ** obj1ExecCost );
witReturnCode witGetOperationObj2AuxCost
(    WitRun * const theWitRun,
     const char * const operationName,
     float * obj2AuxCost );
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 107.
- Concerning `witGetOperationFssExecVol` — The WitRun must be in a postprocessed state. See "The State of a WitRun" on page 134.
- Also concerning `witGetOperationFssExecVol` — The current implosion solution must not be user-specified. See "User-Specified Solution" on page 36
- 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.

Error Conditions

- `consumingOperationName` does not match the operationName of an existing operation.

Example

```
int nBomEntries;
witGetOperationNBomEntries(theWitRun, "op1",
                           &nBomEntries );
printf( "op1 has %d BOM entries\n", nBomEntries );
```

The number of BOM entries for operation "op1" is obtained and displayed.

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.

Error Conditions

- `producingOperationName` does not match the operationName of an existing operation.

Example

```
int nBopEntries;
witGetOperationNBopEntries(theWitRun, "op1",
                           &nBopEntries );
printf( "op1 has %d BOP entries\n", nBopEntries );
```

The number of BOP entries for operation "op1" is obtained and displayed.

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 107.

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 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 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 witSetOperationObj1ExecCost
(    WitRun * const theWitRun,
     const char * const operationName,
     const float * obj1ExecCost );
witReturnCode witSetOperationObj2AuxCost
(    WitRun * const theWitRun,
     const char * const operationName,
     const float obj2AuxCost );
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 134.

Error conditions

- An operation with the specified operationName has not been previously defined.

- Any violations of the requirements listed in “Operation Attributes” on page 107.
- A bound set vector value which does not satisfy the requirements for a bound set. See “API Bound Set Definition” on page 134.

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 113.

- The object iteration process must not be active.

Error conditions

- Any violations of the requirements listed in “BOM Entry Attributes” on page 113.

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 113.

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. 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.
2. 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 204 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.

Error Conditions

- A BOM entry with the specified consumingOperationName and bomEntryIndex must have been previously defined. See also “Error Conditions” on page 214.

Example

```
int nBom, nSubBom, i;
witGetOperationNBomEntries(theWitRun, "oper1", &nBom );
for ( i=0; i<nBom; i++ ) {
     witGetBomEntryNSubsBomEntries( theWitRun,"oper1",i,
                 &nSubBom);
printf("BOM Entry %d has %d substitutes\n",i,nSubBom);
}
```

The number of BOM entries for “oper1” is obtained. Then the number of substitutes for each original BOM entry is obtained and printed.

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 113.

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. Setting some attributes will cause the state to change. See “The State of a WitRun” on page 134.

Error conditions

- A BOM entry with the specified consumingOperationName and bomEntryIndex must have been previously defined. See also “Error Conditions” on page 214.
- Any violations of the requirements listed in “BOM Entry Attributes” on page 113.

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 119.

Error conditions

- Any violations of the requirements listed in “Substitute BOM Entry Attributes” on page 119.
- 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 119.

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 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 witGetSubsBomEntryObj1SubCost
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const int subsBomEntryIndex,
     float * * obj1SubCost );
witReturnCode witGetSubsBomEntryObj2SubPenalty
(    WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const int subsBomEntryIndex,
     float * obj2SubPenalty );
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 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. Concerning `witGetSubsBomEntryConsumedPartName`, `witGetSubsBomEntrySubVol` — It is the responsibility of the application to free the returned vector.
2. Concerning `witGetSubsBomEntryFssSubVol` — this function causes WIT to compute the FSS if needed.
3. 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 204 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 134.
- Also concerning `witGetSubsBomEntryFssSubVol` — The current implosion solution must not be user-specified. See “User-Specified Solution” on page 36

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 119.

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 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 witSetSubsBomEntryObj1SubCost
(   WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const int subsBomEntryIndex,
     const float * const obj1SubCost );
witReturnCode witSetSubsBomEntryObj2SubPenalty
(   WitRun * const theWitRun,
     const char * const consumingOperationName,
     const int bomEntryIndex,
     const int subsBomEntryIndex,
     const float obj2SubPenalty );
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 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. Setting some attributes will cause the state to change. See “The State of a WitRun” on page 134.

Error conditions

- A substitute BOM entry with the specified consumingOperationName, bomEntryIndex, and subsBomEntryIndex must have been previously defined. See also “Error Conditions” on page 224.
- Any violations of the requirements listed in “Substitute BOM Entry Attributes” on page 119.
- `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 124.

Error conditions

- Any violations of the requirements listed in “BOP Entry Attributes” on page 124.
- 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 124.

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 witGetBopEntryPipShare
(    WitRun * const theWitRun,
     const char * const producingOperationName,
     const int bopEntryIndex,
     float ** pipShare );
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. 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.

2. 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 204 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 ≤ 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 124.

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 witSetBopEntryPipShare
(    WitRun * const theWitRun,
     const char * const producingOperationName,
     const int bopEntryIndex,
     const float * pipShare );
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. Setting some attributes will cause the state to change.   See “The State of a WitRun” on page 134.

Error conditions

- A BOP entry with the specified producingOperationName and bopEntryIndex must have been previously defined. See also “Error conditions” on page 233.
- Any violations of the requirements listed in “BOP Entry Attributes” on page 124.

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

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 54. 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 191.
- 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 65. This function works in the same way as `witEqHeurAlloc`, except as indicated below. See "witEqHeurAlloc" on page 237.

`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 237.

Usage Notes

See "witEqHeurAlloc" on page 237.

Error conditions

- See "witEqHeurAlloc" on page 237.
- 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 46.

`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 36. 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 4 on page 135 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 52.

`theWitRun`

Identifies the WIT problem to be used by this function.

Error conditions

- Heuristic allocation must be active.

Example

See “Example” on page 247.

witHeurImplode

```
witReturnCode witHeurImplode
( WitRun  * const theWitRun );
```

Description

Performs the heuristic implosion. See “What Heuristic Implosion Does” 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 52. 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 191.
- `shipPeriod` must be in the range:

  0 ≤ `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 65. This function works in the same way as `witIncHeurAlloc`, except as indicated below. See “witIncHeurAlloc” on page 246.

`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 246.

Error conditions

- See “witIncHeurAlloc” on page 246
- 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 66.

`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 “What Optimizing Implosion Does” 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` and `witOptPreprocess` has not been invoked, `witOptImplode` will perform preprocessing.
3. OSL messages are written to the file defined by `oslMesgFileName`. This file is opened with a FORTRAN open statement with `status=unknown`. On most platforms this means that an existing file will be overwritten. The file is closed by `witOptImplode` before returning.

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.
- objChoice must not be 0.
- 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 52.

`theWitRun`

Identifies the WIT problem to be used by this function.

Error conditions

- Heuristic allocation must be active.

Example

See “Example” on page 247.

witStartHeurAlloc

```
witReturnCode witStartHeurAlloc
( WitRun  * const theWitRun );
```

Description

Initiates heuristic allocation. See “Heuristic Allocation” on page 52.

`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 247.

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 `WitSTDOUT` 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.

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. If this function is not printing any messages, then check to be sure informational message printing has not been turned off by using `witSetMesgTimesPrint`.

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 `WitSTDIN` is specified, then file is read from `stdin`.

Usage notes

1. See “Input Data File” on page 341 of Appendix B for the format of this file.
2. If `WitSTDIN` is not the `fileName`, then 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 invalid data.
- If the file adds any data objects, the object iteration process must not be active.

Example

```
witReturnCode rc;

rc = witReadData(theWitRun,"main.data");
rc = witSetObj2Winv(theWitRun,2.0);
rc = witReadData(theWitRun,"supply.data");
```

A WIT problem is built up by reading the main data from main.data, setting obj2Winv to 2.0, 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 `WitSTDOUT` 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).

`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 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. When the fileFormat is `WitBSV`, then the format of the output record can be found in “Critical Parts List Output File” on page 367 of Appendix B.
4. When the fileFormat is `WitCSV` the same information is written except values are separated by commas and the message number is WIT0380I.
5. 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`.
6. If this function is not printing any messages, then check to be sure informational message printing has not been turned off by using `witSetMesgTimesPrint`.
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 268 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 `WitSTDOUT` 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.

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 82.) 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 321)

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 `WitSTDOUT` 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 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. See “Execution Schedule Output File” on page 360 of Appendix B for the file formats.
4. 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.
5. The messages written by `witWriteExecSched` do not wrap to the next line after writing 80 characters. These messages extend beyond 80 characters.
6. 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.
7. 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`.
8. If this function is not printing any messages, then check to be sure informational message printing has not been turned off by using `witSetMesgTimesPrint`.

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 `WitSTDOUT` 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 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. If the file format is `WitBSV`, then the format of the output file is in “Requirements Schedule Output File” on page 365 of Appendix B.
4. 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.
5. The required volume written is the `reqVol` attribute of a part.
6. The messages written by `witWriteReqSched` 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 is WIT0379I.
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`.
9. If this function is not printing any messages, then check to be sure informational message printing has not been turned off by using `witSetMesgTimesPrint`.

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 `WitSTDOUT` 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 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. A shipment volume of zero is not written to the file.
4. If the `fileFormat` is `WitBSV`, the format of the output record is found in "Shipment Schedule Output File" on page 363 in 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 messages written by `witWriteShipSched` 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 is WIT0380I.
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`.
9. If this function is not printing any messages, then check to be sure informational message printing has not been turned off by using `witSetMesgTimesPrint`.

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 56.)

`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 194. 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 dstWitRun,
  WitRun  * const srcWitRun );
```

Description

This function copies the input data in `srcWitRun` to `dstWitRun`. 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
- oslMesgFileName
- 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.)

`dstWitRun`

The destination WitRun, where the input data is being copied into.

`srcWitRun`

The source WitRun, where the input data is being copied from.

Usage notes

1. This function begins by invoking the internal equivalent of `witInitialize` on `dstWitRun`, which discards all previous data in `dstWitRun`, both input data and output data, except the message attributes, which are preserved. This puts `dstWitRun` into an unaccelerated, unpostprocessed state.

2. If `dstWitRun` and `srcWitRun` are actually the same WitRun, this function does nothing and the input data, output data and state of `dstWitRun` are preserved.

Exceptions to General Error Conditions

- Need not be preceeded by a call to `witInitialize`.

Example

```
WitRun *dstWitRun,  *srcWitRun;

witNewRun( &dstWitRun );
witNewRun( &srcWitRun );

witSetMesgFileName( dstWitRun, WitTRUE, "dst.out");
witSetMesgFileName( srcWitRun, WitTRUE, "src.out);

witInitialize(srcWitRun);
witReadData(srcWitRun,"wit.data");

witCopyData(dstWitRun,srcWitRun);
```

`dstWitRun` and `srcWitRun` now contain the same input data. `dstWitRun` messages are written to file `dst.out` and `srcWitRun` messages are written to `src.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, WIT 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 directly free memory that was allocated by WIT, but in some situations it may be necessary for WIT to free the memory. This situation arises when an application is using a different runtime library than the one being used by WIT.

`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.

Error conditions

- objChoice must not be 0.

Example

```
WitRun * theWitRun;
witGeneratePriorities( theWitRun );
```

The demand priority data is replaced with priorities computed from the objective function attribute data.

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` (of course).

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`.

witOptPreprocess

```
witReturnCode witOptPreprocess
( WitRun  * const theWitRun );
```

Description:

This function processes the data before `witOptImplode` and after the input functions. It is not necessary for the application to explicitly invoke this function.

`theWitRun`

Identifies the WIT problem to be used by this function.

Usage notes:

1. This preprocessing will cause `witDisplayData` to provide additional information.
2. If the preprocessing done by `witPreprocess` has not yet been done, then a call to `witOptPreprocess` will cause this preprocessing to be done.
3. If `witOptPreprocess` preprocessing has not been done when `witOptImplode` is invoked, then `witOptImplode` will perform this preprocessing. This is why it is not necessary for the application to explicitly invoke this function.

Error conditions

- The problem must contain at least one part.

Example:

```
witReturnCode rc;
rc = witOptPreprocess(theWitRun);
```

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 134.

`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 134. 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 124 .

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 31.) 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.

Object Iteration Functions

witAdvanceObjItr

```
witReturnCode witAdvanceObjItr (
   WitRun * const theWitRun);
```

Description

This function advances the object iteration process. (See “Object Iteration” on page 34.)

`theWitRun`

Identifies the WIT problem to be used by this function.

Example

See “Example of Application Code that Uses Object Iteration Functions” on page 289.

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 289.

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 289.

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 289.

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 289.

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 289.

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 289.

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 289.

witResetObjItr

```
witReturnCode witResetObjItr (
   WitRun * const theWitRun);
```

Description

This function puts the object iteration process into its inactive state. (See "Object Iteration" on page 34.)

`theWitRun`

Identifies the WIT problem to be used by this function.

Example

See "Example of Application Code that Uses Object Iteration Functions" on page 289.

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 42.

`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 191.
- `shipPeriod` must be in the range:

  0 ≤ `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 42.

`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 291.

witClearPipSeq

```
witReturnCode witClearPipSeq
( WitRun  * const theWitRun );
```

Description

Clears (i.e., empties) the PIP shipment sequence. See “Post-Implosion Pegging” on page 42.

`theWitRun`

Identifies the WIT problem to be used by this function.

Example

See “witAppendToPipSeq” on page 291.

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);
```

Description

Retrieves the post-implosion consVol pegging associated with a specific demand in a specific shipment period. See “Post-Implosion Pegging” on page 42. 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 `(*peggedConsVolList)[i]` is a pegged consVol. The pegged consVol is the portion of the consVol 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 consVol pegging is to be returned (i.e., the partName of the demanded part).

`demandName`

The demandName for the demand whose consVol pegging is to be returned.

`shipPeriod`

The shipment period whose consVol 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`.

`peggedConsVolList`

On return, for `0 ≤ i < (*lenLists), (*peggedConsVolList)[i]` is the pegged consVol 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 191.
- `shipPeriod` must be in the range:

  $0 \leq$ `shipPeriod` < nPeriods
- The pipExists global attribute must be TRUE.

Example

`witGetDemandConsVolPip` works similarly to `witGetDemandExecVolPip`. See “witGetDemandExecVolPip” on page 297 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 42. 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 191.
- `shipPeriod` must be in the range:

  0 ≤ `shipPeriod` < nPeriods
- The pipExists global attribute must be TRUE.

Example

```
void prtExecVolPip (
      WitRun *     theWitRun,
      const char * partName,
      const char * demandName,
      int          shipPer)
   {
   int      lenLists;
   char * * operationNameList;
   int *    execPeriodList;
   float *  peggedExecVolList;
   int      theIdx;

   witGetDemandExecVolPip (
      theWitRun,
      partName,
      demandName,
      shipPer,
      & lenLists,
      & operationNameList,
      & execPeriodList,
      & peggedExecVolList);

   printf (
      "Post-Implosion ExecVol Pegging:\n"
      "   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:

```
prtExecVolPip (theWitRun, "Prod12", "Cust7", 2);
```

might generate the following output:

```
Post-Implosion ExecVol Pegging:
   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
```

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);
```

Description

Retrieves the post-implosion prodVol pegging associated with a specific demand in a specific shipment period. See “Post-Implosion Pegging” on page 42. 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 `(*peggedProdVolList)[i]` is a pegged prodVol. The pegged prodVol is the portion of the prodVol 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 prodVol pegging is to be returned (i.e., the partName of the demanded part).

`demandName`

The demandName for the demand whose prodVol pegging is to be returned.

`shipPeriod`

The shipment period whose prodVol 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`.

`peggedProdVolList`

On return, for `0 ≤ i < (*lenLists)`, `(*peggedProdVolList)[i]` is the pegged prodVol 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 191.
- `shipPeriod` must be in the range:

  0 ≤ `shipPeriod` < nPeriods
- The pipExists global attribute must be TRUE.

Example

`witGetDemandProdVolPip` works similarly to `witGetDemandExecVolPip`. See “witGetDemandExecVolPip” on page 297 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 42. 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 191.
- `shipPeriod` must be in the range:

  0 ≤ `shipPeriod` < nPeriods
- The pipExists global attribute must be TRUE.

Example

`witGetDemandSubVolPip` works similarly to `witGetDemandExecVolPip`. See “witGetDemandExecVolPip” on page 297 for an example of its use.

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);
```

Description

Retrieves the post-implosion sideVol pegging associated with a specific demand in a specific shipment period. See “Post-Implosion Pegging” on page 42. 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 `(*peggedSideVolList)[i]` is a pegged sideVol. The pegged sideVol is the portion of the sideVol 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 sideVol pegging is to be returned (i.e., the partName of the demanded part).

`demandName`

The demandName for the demand whose sideVol pegging is to be returned.

`shipPeriod`

The shipment period whose sideVol 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`.

`peggedSideVolList`

On return, for `0 ≤ i < (*lenLists), (*peggedSideVolList)[i]` is the pegged sideVol 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 191.
- `shipPeriod` must be in the range:

  0 ≤ `shipPeriod` < nPeriods
- The pipExists global attribute must be TRUE.

Example

`witGetDemandSideVolPip` works similarly to `witGetDemandExecVolPip`. See “witGetDemandExecVolPip” on page 297 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 42. 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 191.
- `shipPeriod` must be in the range:

  0 ≤ `shipPeriod` < nPeriods
- The pipExists global attribute must be TRUE.

Example

`witGetDemandSupplyVolPip` works similarly to `witGetDemandExecVolPip`. See “witGetDemandExecVolPip” on page 297 for an example of its use.

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 42.

`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 291.

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 132.

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 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 OSL message 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, "w" );
```

The access mode for opening the WIT message file is changed to `"w"`.

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 `WitSTDOUT` is specified then 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 `WitSTDOUT`.
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`, 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 severe 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.

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 77, 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 82 and “witWriteData” on page 258.)

APPENDIX A API Sample Code

Sample 1

```
/******************************************************************************
 *
 * Sample WIT API Program
 *
 * Licensed Materials - Property of IBM
 * 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 );
  witSetObjChoice   ( theWitRun, 1 );
  witSetTitle       ( theWitRun, “quote mark: \” back slash: \\” );

  /* Create objects */

  /* Create part A, operation A, and connecting BOP */
  {
  float obj1StockCost[] = { 50., 50., 50., 50. };
  float obj1ScrapCost[] = { 50., 50., 50., 50. };
  witAddPartWithOperation( theWitRun, “A” );
  witSetPartObj1StockCost( theWitRun, “A”, obj1StockCost );
  witSetPartObj1ScrapCost( theWitRun, “A”, obj1ScrapCost );
  }

  /* Create part B */
  {
  float obj1StockCost[] = {  1.,   1.,  1.,  1. };
  float obj1ScrapCost[] = { 10.,  10., 10., 10. };
  float supplyVol    [] = {  0., 100.,  0., 50. };
  witAddPart             ( theWitRun, “B”, WitMATERIAL );
  witSetPartSupplyVol    ( theWitRun, “B”, supplyVol     );
  witSetPartObj1StockCost( theWitRun, “B”, obj1StockCost );
  witSetPartObj1ScrapCost( theWitRun, “B”, obj1ScrapCost );
  }

  /* Create capacity C */
  {
  float supplyVol    [] = {  30.,  30.,  30.,  30. };
  witAddPart             ( theWitRun, “C”, WitCAPACITY );
  witSetPartSupplyVol    ( theWitRun, “C”, supplyVol     );
  }

  /* Create part E */
  {
  float obj1StockCost[] = {   1.,   1.,   1.,   1. };
  float obj1ScrapCost[] = {  10.,  10.,  10.,  10. };
  float supplyVol    [] = {  25.,  25.,  25.,  25. };
  witAddPart             ( theWitRun, “E”, WitMATERIAL );
  witSetPartSupplyVol    ( theWitRun, “E”, supplyVol     );
  witSetPartObj1StockCost( theWitRun, “E”, obj1StockCost );
  witSetPartObj1ScrapCost( theWitRun, “E”, obj1ScrapCost );
  }

  /* 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 obj1ShipReward   [] = { 1000., 1000., 1000., 1000. };
float obj1CumShipReward[] = {   10.,   10.,   10.,   10. };
witSetDemandDemandVol        ( theWitRun, “A”, “F”, demandVol );
witSetDemandObj1ShipReward   ( theWitRun, “A”, “F”, obj1ShipReward );
witSetDemandObj1CumShipReward( theWitRun, “A”, “F”, obj1CumShipReward );
}

/* 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
 *
 * Licensed Materials - Property of IBM
 * 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
 *
 * Licensed Materials - Property of IBM
 * 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 );
   witSetObjChoice( theWitRun, 1 );

   /*
    * Add the parts and operations
    */
   witAddPart( theWitRun, “PartA”, WitMATERIAL );
   {
     float stockCost[] = {   .01,   .01, .01, .01, .01 };
     witSetPartObj1StockCost( 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 );
}
```

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 a “release specification” followed by zero or more “commands”. The valid commands are:

- “add” command
- “set” command
- “read” command

Each command ends with a semicolon.

The release specification simply indicates which release of WIT the file is targeted for. An example of a release specification is as follows:

```
release "5.0";
```

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> ::
    <release_specification> <command_list>

<release_specification> ::
    release <release_num>;

<release_num> ::
    "4.0"
  | "4.1"

<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. An ATTRIBUTE_NAME can be the name of any input attribute listed in chapter 2, except for the following:
    - appData
    - fssShipVol
    - optInitMethod
    - oslMesgFileName
    - Any attribute listed as an “immutable input attribute” in Chapter 2
2. The ATTRIBUTE_NAME must match the object type being set or added, e.g., supplyVol is only allowed for a part.

3. 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.)
4. The type and value of an <attribute_value> must be appropriate to the attribute, as defined in Chapter 2. E.g., unitCost must be a float ≥ 0.0.
5. The number of <vector_value>s specified in a `dense` <vector_format> must be equal to nPeriods.
6. A <period>, t, must satisfy $0 \leq t <$ nPeriods.
7. The same <period> cannot be specified more than once within the same `sparse` <vector_format>.
8. A <bound_type> (i.e., hardLB, softLB, hardUB) must not be specified more than once within the same <bound_set_format>.
9. 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.
10. The STRING given as a <file_name> will be interpreted as the name of an input data file.
11. 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.
12. 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 132.)

TABLE 5 **Arguments for the "add" and "set" Commands**

Object Type	Arguments for "add"	Arguments for "set"
part	partName partCategory	partName
demand	demandedPartName demandName	demandedPartName demandName
operation	operationName	operationName
bomEntry	consumingOperationName consumedPartName	consumingOperationName bomEntryIndex

TABLE 5 **Arguments for the “add” and “set” Commands**

Object Type	Arguments for “add”	Arguments for “set”
subEntry	consumingOperationName bomEntryIndex consumedPartName	consumingOperationName bomEntryIndex subsBomEntryIndex
bopEntry	producingOperationName producedPartName	producingOperationName bopEntryIndex
problem	“add” not allowed	no arguments
partWithOperation	partName	“set” not allowed

13. The object identified in a “set” command must already exist.

14. If an argument in a command identifies another object (e.g., consumedPartName), that object must already exist.

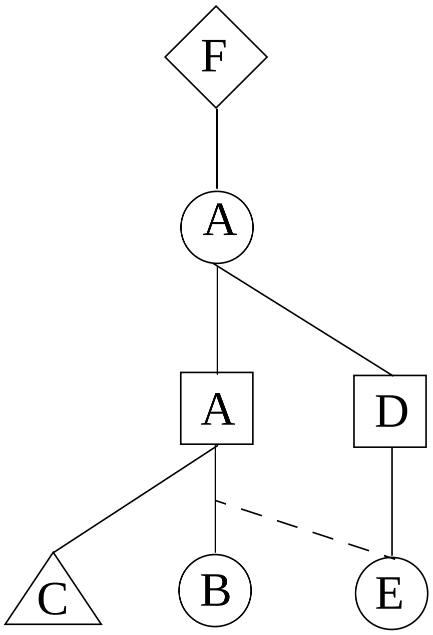
15. The various constraints on objects and attributes defined in Chapter 2 all apply, e.g., a part’s partName must be unique.

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 18 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
//--------------------------------------------------------

release "4.1";

//--------------------------------------------------------
// Setting problem attributes.
//--------------------------------------------------------

set problem
   nPeriods 4
   execEmptyBom true
   objChoice 1
   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"
   obj1StockCost single (5.0)   // attributes of part A
   obj1ScrapCost single (50.0);

add part "B" material
   obj1StockCost single (1.0)
   obj1ScrapCost single (10.0);

add part "C" capacity
   supplyVol single (30.0);

add part "E" material
   obj1StockCost single (1.0)
   obj1ScrapCost 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)
   obj1ShipReward single (1000.0)
   obj1CumShipReward 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".
//--------------------------------------------------------

release "4.1";

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 6 **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	
osl_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
osl_ofname	Name of the OSL 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
opt	Performs an Optimization implosion.
heur	Performs a Heuristic implosion
mrp	Performs the 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 "objChoice" on page 87.) 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 31. 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 ≤ 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 7 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 108.	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 8 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 123	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 9 **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 106.	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 10 **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 100.	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 37 for an explanation of this feature. Each line of this file is in the following column format:

TABLE 11 **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.

“They reel to-and-fro, and stagger like a
drunken man: and are at their wit’s end.”

Prayer Book (1662)