CUSTOMIZING XCONQ (Made Somewhat Simple)

Stan Shebs

Department of Computer Science
University of Utah

ABSTRACT

For advanced players who desire new challenges, *xconq* provides several customization mechanisms. It is possible to build new maps, scenarios, and historical periods. This document describes the customization process in excruciating detail. The reader is assumed to be familiar with *xconq* and its basic documentation.

INTRODUCTION

Xconq has been designed to facilitate the construction of new maps, scenarios, and historical periods. All these additions happen through a single mechanism, the *mapfile*, which comes in several parts, and may also load other mapfiles. Saved games are very detailed mapfiles; less detailed mapfiles are more suitable as scenarios. Maps and period descriptions are mapfiles with most sections missing.

Mapfiles are created either by text editing, by *xconq*, or by a combination of both. Since they are mostly read by variations on C's *scanf* function, the format is rather rigid; numbers may not be omitted, no blank lines are allowed (but whitespace between numbers on a line is OK), and mistakes will not always be recognized. The one general exception is the period description, which is read by a lexical analyzer, which allows free-format input. Other more specific exceptions will be noted below.

MAPFILES

A mapfile is a normal ASCII text file. It begins with a header line of the form

Xconq *n [+-][+-][+-][+-][+-][+-] description[;]*

where *n* is the number of additional mapfiles to be loaded (up to 16 or so in all), each + or - indicates the presence or absence of a particular section, and the *description* is free-form text (remainder of the line) describing the mapfile. Note that that all reading/writing in the mapfile is case-sensitive, including the header words. The sections are for version, period, map, globals, sides, and units, in that order. Most sections vary in size, and some vary also in the detail of the information they contain. Most include extension flags, which are reserved for future expansion of mapfiles.

Additional notes may be included by terminating the *description* with a semicolon (with NO whitespace following). In such a case, the following text lines will skipped over until a period "." is seen by itself on a line.

Following the header are the names of the *n* mapfiles, one to a line. Each will be loaded into *xconq* before the rest of the current mapfile. This process is recursive (at least in theory). Any conflicts are resolved by simple overwriting. Note that this presents the possibility of catastrophic conflicts if the designer is not careful to, say, match units and periods. A file with a given name will be loaded only once; a second attempt will result in a warning message, and the program will continue on.

The version section is quite simple, since it is one line; exactly the version string displayed when *xconq* starts up. This is typically only needed for saved games, but might be convenient while *xconq* versions are changing incompatibly. When this section is loaded, the version is merely compared with the internal version number, and *xconq* fails if they do not match.

The period section is a program in a simple postfix language, which when executed fills in the period description.

The map section includes the world dimensions and terrain. It may also include information about countries at some future date.

The globals section includes any global values, such as the current turn, and also the win/lose conditions for a scenario.

The sides section defines the sides in a game. Required info is little more than the name of the side, but a detailed section may also include the area of the map seen by that side, and other data.

Finally, the units sections lists a number of units in the game. Again, detail can range from type, name, and position to complete details on its orders.

Each of these sections is entirely optional - *xconq* can synthesize its own replacements for each. For instance, a mapfile could specify a map and two sides, or a period, or an entire game, by including the appropriate sections. The sections do interact somewhat, with dependencies as follows:

Maps, win/lose conditions in globals, detailed sides, and units depend on period.

Detailed and/or non-neutral units depend on sides.

The following sections discuss each part of a mapfile in more detail. The version secton has already been covered sufficiently. Since the period defines what may appear in all the other sections, I shall begin with it.

PERIODS

Historical periods are the most complicated way to customize xconq, and can be difficult to get right. There are many hundreds of numbers, each of which must be in balance with every other. At best, a mistake will result in a period whose winning strategies are quite simple; at worst, *xconq* will hang or core dump. (*xconq* will make some efforts to check the numbers.) Although most historical periods are defined in the period sections of mapfiles, one period is compiled into *xconq* (usually a version of WW II, since it is most familiar). Compiling other periods (using the utility*per2c*) is straightforw ard, and there is no problem with making unusual default periods for the program, but in practice, periods do not take very long to read.

The header of the period section has the form

Period *extension*

where *extension* is a flag indicating whether the period info is to replace (= 0) or add to (= 1) any existing period info.

The remainder of the description is in a postfix language similar to Forth or Postscript(tm), but even simpler. *xconq* includes an interpreter for this language, and program execution results in a filled-in period description. There is a compiler, called *per2c*; it is used to produce a C version of the period definition which is then compiled into *xconq*.

The objects of this language are quite simple. There are only integers, strings, and vectors of integers. Integers and strings have direct representation as tokens, as do symbols, while integer vectors must be constructed using square brackets or one of the predefined vector-returning words. Tokens are always separated by whitespace, which means whitespace may *never* be part of a token (backslash may be used for this purpose at some future date). Thus, the syntax for a token is fairly simple:

token -> [+-]*[0-9][0-9]*[%]* | "[~"]*" | [~"+−0-9].*

In other words, numbers are digits optionally preceded by +/- and followed by %, while strings are anything enclosed in double quotes, and symbols are anything else that is not white space. A semicolon is the comment character; the characters from the semicolon to the end of the line are discarded. Numbers are limited to the typical range for signed shorts; -32768 to 32767. The percent sign % is just for readability (since many numbers are percentages), and its presence or omission has no effect on anything.

Numbers and strings merely get pushed on the stack, while symbols are assumed to be defined words. The reader looks them up in the dictionary and executes the associated C function, if one is defined (otherwise, the symbol gets pushed). Nearly all words take a fixed number of arguments from the stack and push

nothing, although there are some exceptions. The program is terminated by the word **end**.

Integer vectors are useful for filling in large parts of arrays. They are individual objects, built using the words **[** and **]**.

Usage of these depends on the word, but typically words with two or more arguments may include two vectors among them. For instance, the line

[1 1 2 8] [i a b B] hp

sets the hit points for four types of units; **1** for unit type **i**, **8** for unit type **B**, and so forth.

The words below generally follow some extremely regular patterns. Most one-argument words define things for the entire period. Most two-argument words define attributes of units or resources, and may be thought of one-dimensional array operations, while three-argument words usually fill in two-dimensional arrays. In the two-argument and three-argument cases, the first argument (deepest in the stack) is the value, while second and third arguments are indices. At present, there are no words with four or more arguments.

One-dimensional operations allow three cases of the four possibilities for vectors and scalars: scalar value and index(set), scalar value/vector index (fill), vector value and index (mapping). Two-dimensional operations offer six of the eight possible combinations (encoded here with S for scalar and V for vector, in the order value,index,index), disallowing only a vector value with either two scalar or two vector indices (VSS and VVV), since these combinations are not very meaningful. The six other combinations are SSS (element set), SVS and SSV (row fill), SVV (array fill), VVS and VSV (row mapping). All of these combinations are useful - see existing periods for examples of their use.

The following descriptions cover all the predefined words of the period language. Most words correspond to period attributes, and thus have an associated default value; when not explicitly mentioned, the default is **0**. For arguments of type *bool*, both **1** and **0** or **true** and **false** are valid (an argument characterized as *bool* may still be a vector of 1s and 0s). Arguments of type *n*, *n%*, and *n.01%* are all just integers.

First, there are some useful words not specific to period definition.

true

false Pushes a **1** and a **0** on the stack, respectively.

[Marks the beginning of a vector. The following objects must all evaluate to numbers only.

] Marks the end of a vector. All numbers going back to the last**[** are popped and collected into a v ector, which is pushed back on the stack.

value name **define**
Define the string *name* to be a word that pushes the object *value* onto the stack. This is especially useful for assigning names to vectors, for instance a vector of all cities or all ships.

print
Print the current contents of the stack onto standard output, surrounded by /* */. Very useful for debugging!

The global period definition words.

string **period-name**
Defines the name of the period that will be displayed on startup. Defaults to **"unspecified"**.

string **font-name**
Defines the name of the font that will be used. This is not needed if bitmaps are being used instead. Defaults to the name of the text font. See below for a discussion of bitmaps and fonts.

The words to define new kinds of units, resources, and terrain all add new words that push the number of their unit, resource, or terrain type onto the stack. The numbering is in order; the first of each type will be numbered **0**. Most of the period definition words use these values as indices when filling up arrays of numbers.

char name string **utype**
Define a type of unit. The arguments are unit character (a string of length one), full name, and a one-line help string. The unit may thereafter be referred to by either its character (as a one-char symbol) or by its name. Things will not work if a blank or any other special characters (such as plus and

minus signs) are used as a unit character. Mixing terrain characters and unit characters is also a bad idea, since both may be used by a graphical display. At present, up to 30 or so unit types may be defined.

char name help **rtype**
Define a type of resource, by both name and character, and supply a help string for it. The characters must be distinct from unit characters, otherwise this word is identical to **utype**.

char name color **ttype**
Define a type of terrain. Unlike the other two definers, the terrain character does *not* become a new word, although the name still does. The character should be defined in the basic *xconq* font (the one named **xconq.onx**, in the X10 interface), if the period is to be used with an interface that needs it (monochrome X only, at this writing). The color may be either approximate ("brown") or exact ("#334455"), but in any case, the interpretation is up to the interface.

u* Push a vector of all unit types in order.

r* Push a vector of all resource types in order.

t* Push a vector of all terrain types in order.

nothing
Push the index of a non-unit onto the stack. This word cannot be used for unit attributes, but it is useful for certain attributes whose values are unit types (**first-unit**, **first-product**). Use of this as an index with any word that sets unit attributes is guaranteed to cause nasty coredumps...

string unit **icon-name**
Set the name of the icon for the given unit type. If undefined, then the unit's character will be used in whichever font is being used for unit icons (which may even be the text font - works but not too attractive). Definition of an icon overrides any character in any font. As with the terrain, the exact interpretation of this word is up to the graphics interface - for instance, the curses interface ignores both the **font-name** and **icon-name** words entirely. In X, the name is the name of a file in the usual bitmap format, as produced by the *bitmap* program. The actual file name is produced by appending **".b"** for X10 bitmaps and **".b11"** for X11 bitmaps. The X11 interface will also look for X10 bitmap files.

terrain **default-terrain**
Set the type of terrain to be substituted while reading a map with incomprehensible terrain characters. Occasionally useful, if the set of terrain types is a subset of the standard set.

Initialization characteristics of a period are used only during synthesis of maps, sides, and units, which happens when they are not supplied from a mapfile. Periods that are used only with complete scenarios need not use any of these words.

n% terrain **min-alt**
Set the minimum percentile of terrain elevations that result in the given terrain type. Together with the other three words following, it is possible to subdivide all the possible altitudes and moisture levels into different kinds of terrain. For instance, desert in the standard period ranges from sea level (**70 desert min-alt**) to high elevations (**93 desert max-alt**) but only in the lowest percentiles of moisture (**0 desert min-wet**, **20 desert max-wet**). It is important that all percentiles be assigned to some terrain type, or the map generator will complain; when designing terrain combinations, it is helpful to make a graph with altitude percentiles 0-100 on one axis and moisture percentiles on the other.

n% terrain **max-alt**
As for **min-alt**, but set the maximum altitude percentile. Defaults to **100**.

n% terrain **min-wet**

n% terrain **max-wet**
Set the minimum and maximum percentiles of world moisture levels that result in the given terrain type. Deserts should range in the low percentiles (0-20), while rain forests should be high (90-100). The minimum defaults to **0**, while the maximum is **100**.

n% **alt-roughness**

Set a rather mysterious number that controls whether a random map tends toward large continents or archipelagos of small islands. It must range between **0** and **100**. Altitude roughness of**0** will result in a map with one large continent, while **100** produces dozens of tiny random islands, and drastic altitude variations from one hex to the next. Defaults to **80**.

n% **wet-roughness**

Set the "moisture roughness", which is like altitude roughness, but affects the distribution of wet and dry areas. Defaults to **70**.

terrain **edge-terrain**

Set the type of terrain to fill in on the northern and southern edges of a map. Best user-friendliness is to have a type that is scarce elsewhere, so the edges are not mistaken for normal terrain.

distance **country-size**

Set the radius of a randomly-placed country, in hexes. The country is always hexagonal in shape, and the center hex is not counted in the radius. The radius should be sufficient to accommodate all the initial units, without crowding them. For radius r, the number of hexes is 3/4*(2r+1)**2. Keep in mind that terrain may reduce the number of available hexes even further. If one type of unit can occupy another, then they are free to be placed in the same hex. Defaults to **3**.

distance **country-min-distance**

distance **country-max-distance**

Set the minimum and maximum distances of country centers from each other, in hexes. These values are sometimes tricky to set properly. If too small, countries will mostly overlap; if too large, then attempts to use small maps will fail; if too close to each other, placements can also fail. Default to **7** and **60** hexes, respectively.

unit **first-unit**

Set the type of unit that player will be started off in. Setting this to **nothing** has the effect of giving every unit in the country to the player at the outset. Production will not be set automatically, so this is not recommended for novices, who tend to find large numbers of units confusing at the outset. Defaults to **nothing**.

unit **first-product**

Set the type of unit that will be built automatically first. A "cheap" (quick to build) type is usually best, although interesting situations could result from, say, the automatic production of one atomic bomb at the outset (note that all sides start out in exactly the same way). Defaults to **nothing**.

n unit **in-country**

Set the number of units of the given type in a player's country. These units are randomly scattered, with some bias towards the middle of the country, and subject to terrain limitations via the **favored** parameter (see below).

n unit **density**

Set the total number of units appearing throughout the map, at the rate of one per ten thousand hexes. The numbers of units appearing in countries is subtracted first, so that the final density of units is independent of the number of players. If this value is nonzero, then at least one unit will appear on the map, even if the map is very small (i.e. the calculation of numbers rounds up not down). Units not assigned to countries to meet the quota set by **in-country** always become neutral.

bool unit **named**

Set a type of unit to get a random name during initialization. The names are usually the names of towns, so this flag should be used judiciously (A battleship named "Wankers Corner" is only briefly amusing!). The value can also be set to **2**, in which case the unit name will be displayed by itself, without side name or unit type name.

name **uname**

Define the string *name* to be a plausible name for random assignment to an initial named unit. The name will be included with any previously defined, including possibly ones in the compiled-in period. This behavior, which is unlike any other period word except for **sname**, is intended to ensure

that each period need not define its own list of names.

clear-unit-names
Reset the list of unit names. This is appropriate if the compiled-in list is completely wrong for the period.

n% terrain unit **favored**
Set the probability of the unit being on the given type of terrain at the outset. The default of **0** is an absolute prohibition against placing the unit on that type of terrain, thus every period must specify at least one non-zero value for some terrain type and some initial unit type. (Note that this does not preclude a unit type with no favored terrain, but it must be able to occupy some other unit already placed. In fact, this is a useful way to force one initial unit to start out inside another.)

This parameter is tricky to use properly, and not very flexible. The problem is a widely differing favored terrains for initial units may be too constraining to work with the typical random map. For instance, very few small countries will include both ice and open sea, or deserts and forests and swamps. Failure to find such combinations will result in games exiting while still initializing, thus frustrating erstwhile players of the period. Best results will be had if the favored terrains are the same for all initial types of units, and the terrain types are common on random maps. (At present, the most-favored, or the lowest-numbered type among equally-favored terrains should be plentiful on the map; this is to get around a bug.)

n **known-radius**
Set the area of the world known about, in those cases where the world is not already known. *n* is measured in hexes, and represents a radius which will be seen around each of the starting units.

bool unit **already-seen**
Set the type of unit that is or is not seen at the outset. This should usually be true of things like cities, independently of their **always-seen** setting.

n% resource unit **stockpile**
Set the percentage of capacity for the given resource that each unit will start out with. Defaults to **100**.

name **sname**
Declare the string *name* to be a plausible name for random assignment to a side. The name will be added to the others already defined, including ones from the compiled-in period.

clear-side-names
Reset the list of side names. This is appropriate if the compiled-in list is completely wrong for the period, but if used, you must supply at least as many side names as there are possible sides (7 or so).

n terrain **inhabitants**
Set the number of inhabitants in each hex of a country with the given type of terrain. The number is relative, and at present is only treated as a boolean value.

n terrain **independence**
Set the "independence" of the inhabitants in the given type of terrain; how they react to enemy units in terms in attrition, supply, etc. [not implemented yet]

The first phase in a turn is devoted to spying. This is the revealing of all or part of a side's unit positions to another randomly-selected side. It is controlled by only two parameters.

n% **spy-chance**
Set the percentage chance of spying occurring on this turn. If the chance is low, then the player doing the spying will get a message, otherwise the display will be silently updated. The player spied upon is never informed. Defaults to **1**.

n% **spy-quality**
Set the percentage of enemy units that will be seen when spying is successful. Defaults to **50** (i.e. on the average about half of the side's units will be seen).

n% **leave-map**
Can the units leave the map?

The second phase in a turn determines any revolts or surrenders, attrition, and disasters. Since these are (usually) rare events, the probabilities are set in one-hundredth percent increments. Revolts happen anywhere, while surrender happens only if enemy units are nearby. Attrition is the loss of single hit points, without actually destroying a unit, while disaster is the complete destruction of the unit (both of these depend on terrain).

Note that with 100 units in play, the lowest possible nonzero chance of 1 for a value still results in an occurence of that sort of disaster every 100 turns or so, so these parameters require a "light touch".

n.01% unit **revolt**
: Set the base chance for the unit to revolt spontaneously in that turn. This chance is reduced by better morale and maybe other things.

n.01% unit **surrender**
: Set the base chance for surrender to some adjacent enemy unit. Each enemy unit present adds to the chance by this amount.

n.01% unit **siege**
: Set the additional chance for surrender when the unit is completely surrounded by enemy units. This is added to the basic surrender chance.

n.01% terrain unit **attrition**
: Set the chance of a unit losing a single hit point while in the given terrain.

n unit **attrition-damage**
: Number of hit points lost when attrition happens. Defaults to **1**. Note that repair is in the following phase, and 1 hp of attrition damage might be repaired immediately, and appear not to have happened.

string unit **attrition-message**
: Set what to say when unit is hit by attrition. Defaults to **"suffers attrition"**. If the string is**""**, then the message will be suppressed entirely.

n.01% terrain unit **disaster**
: Set the chance of completely losing the unit while in the given terrain. Accidents should be restricted to definite hazardous situations, to go along with movement constraints - for instance, carriers in shallow water should move more slowly and have a nonzero accident rate. See random movement for another way to achieve similar effects.

string unit **disaster-message**
: Set what to say when unit is lost in a disaster. Defaults to **"has met with disaster"**.

The next phase of a turn handles creation of new units and repair of damaged units. Units can only be created by certain other kinds of units, limited both by time and raw materials. Also there are startup and research times.

n unit2 unit **make**
: Set the time in turns needed for a unit of type *unit* to build one unit of type *unit2*, assuming sufficient resources to do so.

bool unit **maker**
: Set the unit type to be a "maker". Makers always build unless explicitly idled, and may move while building. If a period starts with no movers, then it needs at least one maker in the country, who will prompt for a unit type at the beginning of a game.

bool unit **occupant-produce**
: If true, then a unit may produce as the occupant of another unit. Default is FALSE. Makers ignore this flag and always produce.

n% unit **startup**
: Set the extra time needed to build the first unit, if the maker was producing something else before. Startup time should be higher for high-tech or large units, for instance to represent tooling or production pipeline startup.

n% unit **research**
: Set the extra time needed for a side to build the very first unit of that type. This time is in addition to

the startup time for the first unit. Long research time is a good way to keep a unit type out of play for awhile.

n% unit2 unit **research-contrib**

Percent of research on *unit2* to count towards *unit*. All research contributions are summed and will never do more than eliminate research on a unit type. Only completed research is counted (i.e., a unit must already have been produced).

n resource unit **to-make**

Set the total amount of a resource type needed to build a unit. This amount is amortized over the normal construction schedule, which means that extra resources are consumed by startup or research times (representing mistakes and experiments).

n unit2 unit **repair**

Set the time needed for the unit type *unit* to repair **repair-scale** hit points of damage to unit type *unit2*. One of the two units must be able to occupy the other; It is also legitimate for a unit to repair itself. If the unit being repaired was crippled, its repair will require the same kinds and amounts of resources that were used to build it.

n **repair-scale**

Set the "scale" of repairs, meaning the overall amount that individual repair rates are relative to. So for instance a repair-scale of 4 means that a repair time of 2 results in the recovery of 2 hp/turn. Defaults to 1.

The supply phase of a turn handles both the production of resources and their distribution via supply lines. Resource production involves a three-dimensional array indexing unit type, resource type, and terrain type, but supply lines are measured only by length and resource type. Supply lines are always interrupted by enemy presence.

n resource unit **produce**

Set the basic amount of each resource produced by each unit in one turn.

n terrain unit **productivity**

Set the percentage productivity of a unit on a type of terrain. This is multiplied with the basic production rate to get actual production, so productivity of **0** completely disables production on that terrain type, and productivity of **100** is yields the maximum rate specified by **produce**.

n resource unit **storage**

Set the unit's capacity to carry each sort of resource. Amount carried does not affect unit's performance. When the value is **0**, displays for that type of unit will not mention this resource type at all.

n resource unit **consume**

Set the amount of resources consumed by the unit in a turn, even if it doesn't move or do anything else. This includes riding as a passenger. This only comes into play if the unit has used less than its base consumption while moving. In other words, the total supply usage for one unit in one turn is max(#moves * to-move, consume). If the unit runs out of a resource that it must consume, it dies due to starvation.

n resource unit **in-length**

n resource unit **out-length**

These two are used together to determine the length (in hexes) of supply lines between units. The given type of resource can only be transferred from unit type A to unit type B if the distance is less than the minimum of the in-length of B and the out-length of A. For instance, the in-length for a fighter's fuel might be 3 hexes, while the out-length of fuel from a city is 4 hexes. If the fighter's out-length is 0, then it will be constantly supplied with fuel when within 3 hexes of a city, but will never transfer any fuel to the city unless it actually lands there. An in- or out-length of **0** means that the two units must be in the same hex, while a negative length disables the automatic transfer completely. Long out-length lines should be used sparingly, since the algorithm uses the out-length to define the radius of search for units to be resupplied. Supply lines are not affected by terrain at present.

*bool unit***consume-as-occupant**

If this is true, than this type of unit does not consume any supplies as long as it is an occupant on

some transport. This is useful for units such as planes which always consume fuel in the air but not on the ground. This defaults to TRUE.

n% unit **survival**
Chance that a unit type can survive on no supplies. The test is made once per turn.

string unit **starve-message**
Set what to say when unit has no more of some supply to consume. Defaults to **"runs out of supplies and dies"**.

The movement phase is the main part of a turn in *xconq*, and the only part involving interaction with players. All combat happens during the movement phase.

n unit **speed**
Set the maximum theoretical speed of a unit, in hexes/turn. If the unit cannot move on any sort of terrain, it will never be prompted about - thus every period should define at least one type of moving unit.

n terrain unit **moves**
Set extra moves used up on each type of terrain. **0** indicates no decrease from theoretical max, **2** indicates a move into that type of terrain uses up 3 moves instead of 1, and **-1** indicates that movement on that type of terrain is not possible. Defaults to **-1**.

n.01% terrain unit **random-move**
Set the randomness of movement of a unit on the terrain. This is different from disaster and attrition, since it is not always fatal, and happens only during attempts to move. However, collisions with other units or with impassable terrain, due to random moves, are always fatal.

bool unit **free-move**
Set whether the unit can move even if there is insufficient movement allowance remaining in this turn. Defaults to **true**. (Most board wargames make this false - if you don't have enough movement points to meet the entry requirement for a hex, that's too bad.) Can be useful to make "double movement phases", if attack time is equal to movement allowance; a unit can only attack units that it is adjacent to at the start of the movement phase.

bool unit **one-move**
Set whether the unit can make exactly one move before dying (appropriate for rockets and other automatic equipment). [not implemented yet]

bool unit **jump-move**
Set whether a unit can jump over another unit to get somewhere. [not implemented yet]

n resource unit **to-move**
Set the amount of resource used by a unit to move one hex. The amount taken is independent of the terrain in the hex. If the unit is out of any movement resource, it is immobilized until it gets more.

Transportation-related parameters. Capacity is measured both by number and volume of occupants. For instance, if you wanted a transport to carry up to 8 infantry and/or armor, but no more than 4 armor units, then capacity for infantry should be 8 and capacity for armor 4, the volumes for each should be 1, while the transport hold-volume should be 8.

n unit2 unit **capacity**
Set the basic carrying capability of a transport type *unit* for its occupants of type *unit2*.

n unit **hold-volume**
Set the volume capacity of a transport. Volume measure is quite arbitrary, and is used only for comparison. The default value of **0** implies infinite capacity, volume-wise.

n unit **volume**
Set the volume of a unit. The volume of a unit may be smaller than its hold-volume, the code will not care about this.

n unit2 unit **enter-time**
Number of moves needed to enter a transport. This is a time measure; extra supplies are not used up.

n unit2 unit **leave-time**
Number of moves needed to leave a transport; similar to **enter-time**.

n% unit2 unit **alter-mobility**
Set the effect of an occupant on the transport's speed as a ratio of the transport's usual speed. Defaults to **100**; smaller values slow the transport, and **0** prevents it from moving entirely. To simplify the code, only the effect of one (randomly chosen) type of occupant has this effect. If a transport has two types of occupants each of which alter its speed differently, the resulting transport speed will be unpredictable. The total slowdown is multiplied by the number of occupants of all types.

Seeing is an important part of *xconq*, and needs parameters to accommodate submarines, radar installations, and Indians hiding in the woods. The visibility of a unit and the intensity of viewing are computed separately, and compared to get the final decision on seeing something. This doesn't allow for much differential between two types of units viewing a third, but that's life. For units seeing things at a distance, the chances are interpolated linearly, from the best conditions (adjacent hex) to worst (maximum range).

bool **all-seen**
If true, then all sides see all of each other's units. If secrecy unneeded (as in a board game), this will speed up the display process somewhat.

n% unit **see-best**
Set the basic chance of one unit seeing any other, under best possible conditions. Defaults to **100**.

n unit **see-range**
Set the maximum distance in hexes at which the unit can see anything. Defaults to **1** (adjacent hexes only).

n% unit **see-worst**
Set the chance of seeing a unit at the maximum range. Defaults to **100**.

n unit **visibility**
Set the basic chance of a unit to be seen. Crippled unit is more visible, in proportion to hp loss. Defaults to **100**.

n% terrain unit **conceal**
Set the percent effect of terrain on seeing the unit. This is subtracted from the basic chance, since it is a "concealment factor".

bool unit **always-seen**
Declare the unit to be of a type that is always seen and up-to-date. This applies only to units whose underlying hexes have been seen. This is useful for units like towns, which are unlikely to disappear secretly.

Combat is part of movement, and has its own large set of parameters. The basic plan of combat is for attackers and defenders to hit each other, then attackers to attempt to capture. Success of a hit attempt depends on a number of attributes, including chances, terrain, and the availability of the correct sort of ammo.

bool unit **multi-part**
Set a unit to be treated as an aggregate of smaller identical units. Affects various things. [not implemented yet]

n unit **hp**
Set the maximum number of hit points for each part of a unit. Defaults to **1**, may never be set any lower.

n unit **crippled**
Set the hit point level below which the unit is considered to be crippled. Below this level, repair and construction ceases, supply production is reduced, maximum speed starts to decrease, and the bridging capability is disabled.

n% unit2 unit **hit**
Base chance of a single attack by the type *unit* hitting the defender *unit2*, assuming the resources are available. If chance to hit is**0**, attack er cannot attack or defend itself.

n% terrain unit **defense**
Set the decreased chance of hitting if the defending type *unit* is in that terrain type. Percentage is subtracted from base chance.

n% **neutrality**
Set the change in defense for neutral units. This is subtracted from chances to hit and capture, but the *n%* can be negative, which would make it harder to hit/capture.

n unit2 unit **damage**
Number of hit points that the defender *unit2* loses when hit by its attacker *unit*.

n **nuke-hit**
Minimum damage for a hit to qualify as a nuclear blast and be displayed appropriately. Default value is **50**.

bool unit **self-destruct**
Declare that unit self-destructs when it attacks. This eliminates some weird messages and hit chances.

bool **counterattack**
When true, combat is two-way; the initiator of an attack is also hit by a counterattack. Otherwise, the defender must wait to get its revenge. Defaults to **true**.

bool **capturemoves**
When true, a captured unit can immediately be moved. If false, then a captured unit can not be moved until the next turn. Default is TRUE.

bool unit **can-counter**
Like **counterattack**, but applies only to particular unit types being attacked. Defaults to **true**.

n% unit2 unit **capture**
Set the base chance of the type *unit* capturing the defender type *unit2*. This is conditional on both attacker and defender surviving initial hits, and is modified by morale and quality of both sides.

bool unit2 unit **bridge**
True if the unit type *unit* can capture another unit *unit2*, even across impassable terrain.

n% unit **changes-side**
Set chance that the given unit will change sides if captured. This is appropriate for units that are primarily hardware or otherwise indifferent to their fate. Units that are captured and do not change sides become prisoners (prisoners are not implemented yet).

n unit2 unit **guard**
Set the number of unit hit points required to garrison or guard a captured type *unit2*, whether or not the captured unit has changed sides (at present, it always does). The hit point loss is permanent.

n% unit **retreat**
Set the base chance that a unit will retreat rather than be hit. This choice depends on ability to move into an adjacent hex and on morale, quality, and fatigue.

n resource unit **hits-with**
Set the amounts of each resource used as ammo by the unit.

n resource unit **hit-by**
Set the amounts of each resource necessary to score a hit on the unit. This is correlated with the previous parameter to decide if right sort of ammo is available for an attack.

n unit2 unit **protect**
Set the level of protection that *unit* offers to *unit2*. Transports protect their occupants by only letting a percentage of hits get through. Occupants protect their transports by reducing the chance of a hit and increasing chance of a counterattack. (The default of **0** implies terrible carnage if a full transport is hit.)

n unit **combat-time**
Set the extra number of moves used by an attack.

string unit **destroy-message**
Set what to say when a unit is killed in combat, as an active verb for what the destroying unit has done to its victim. Defaults to **"destroys"**.

General characteristics are not really classifiable anywhere else.

n unit **territory**
Set the territorial value of a unit. Primarily used by machine players and win/lose conditions.

n unit **max-quality**
Set the maximum quality achievable by a unit.

n% unit **veteran**
Set the effect of one point of quality on hit and capture chances.

n unit **max-morale**
Set the maximum morale to which a unit can rise.

n% unit **control**
Set the chance of a unit obeying its orders. Defaults to **100**. When the unit does not obey orders, it makes a decision using the machine players' algorithm.

bool unit **can-disband**
Set whether a 'D' disband command can be used to get rid of a unit. It should not be possible to disband a city, for instance, to eliminate it as a strategic target. Note that the default of **0** effectively disables the disbanding command entirely.

n% **efficiency**
Units disbanded in a transport can have the resources used to build them reclaimed - this parameter sets the percentage that is actually obtained.

bool unit **neutral**
Set to **true** if unit can exist as a neutral. If **false**, then anything that would cause the unit to become neutral (revolt, surrender of owner) has the effect of removing it instead. Defaults to **false**.

Miscellaneous words.

n **hostility**
Set the level of hostility exhibited by a population toward a unit from some other side. [not implemented yet]

begin{notes}
Declare the beginning of the designer's notes. This word kicks in a special reader that absorbs all lines until it sees the line "**end{notes}**". The intervening lines are saved as period notes and listed out in **"parms.xconq"**. The notes should rationalize the design and discuss features of special interest to the player.

end Marks the end of the period description.

Nearly all the elementary programming errors are checked, such as stack over/underflow, and as many of the period parameters as possible will be checked, although there is plenty of room for subtle loopholes. You should think carefully about the consequences of each parameter, being particularly sensitive to degenerate winning strategies. Most common are units that are too powerful, or that are built so quickly that they overwhelm any opposition. The players should always be a little "hungry"; not able to get quite as much of units or resources as they would really like.

Although there are many interesting possibilities inherent in this period description language, you should avoid making the period too complex to be humanly playable. The compiled form of the period description can involve over 16,000 individually settable numbers, each with an expected range of perhaps 100 distinct values. It is clearly possible to spend many years exploring a single set of these numbers. For more playable and enjoyable games, either pick a single aspect to treat in detail, or else do all aspects in a simplified way. Aspects could include exploration, logistics, naval operations, "shoot-em-up", renditions of familiar board games or even some team sports (rugby for instance). Another thing to keep in mind is that the introduction of a new type may have far-reaching consequences - a new unit type will need its interactions with *all* other unit types defined. One approach is to introduce a new type as a slight modification of

an existing type, then to share most of the definitions.

Something else to keep in mind is that the period parameters have been chosen for their ability to combine in interesting ways, rather than for obvious usefulness. For example, past startup, the production rate for units is constant and unending. But suppose you want to put a limit on the numbers of that type of unit? One way is to define a resource that is essential for construction of that type, let the builder have an initial supply, but provide no way to get more of that resource. When it runs out, no more units! Another trick is to motivate an activity by making it a prerequisite to the basic builtin goal of defeating the other player. The age of discovery worked this way. The kings of that time weren't interested in new lands per se; they wanted exploitable possessions that could be used to get gold to buy armies big enough to defeat their neighbors. The period language could describe this situation almost exactly, by making gold a resource obtainable only by the capture of neutral mines thinly scattered over the map. Be inventive! Studying the predefined periods should reveal a number of tricks.

Completely new periods usually have a number of bugs. The tools are rather limited, but then most of the bugs are fairly obvious. The**print** w ord is useful for examining the stack, and a number of errors (such as stack overflow/underflow) have messages. Finding out where the problem occurred requires the use of the *xconq* debugging flag **-D**, which has the effect of listing out each period token as it is read. This can also be used with the period compiler, which starts up faster; invoke it as **"per2c -D <newperiod.per"**. The most serious problems with periods are play balance issues. Some can be found out by watching a machine player, since its decisions are based on perceived values of the units. The most subtle bugs can only be uncovered by extensive play interspersed with judicious alteration of parameters. I find it useful to play for a while, then go over all the period parameters, thus avoiding tweaking one parameter only to find that it results in another being inconsistent. Parameters interact in many ways - you should keep this in mind when experimenting.

MAPS

Maps alone are the easiest thing to customize. You can either do ordinary text editing to acquire the map data, then add the headers, or use the online terrain editing capability of *xconq* and save the terrain alone.

The map header has the form

Map *width height scale seen extension*

where *width* and *height* are the size of the map, while the *scale* is the width of a single hex in km (measured between parallel faces). The*seen* flag is **0** if the map is not kno wn to the players, and **1** if it is (this is like using **"-v"** on the command line). The *extension* is an extension flag, with a value of **0** for the normal map. Other values are reserved for extensions.

The preferred way to build a map is to use the **"-B"** option of *xconq*. This enables the space bar to "paint" terrain onto the map. The "brush" is defined by using the terrain characters as commands (they will override any normal commands on the same letters). The argument is also useful - if positive, it is the radius of the area to be painted with the given terrain type, if negative, it is the length of a horizontal bar. Thus, in the standard period, the sequence **"999. "** will convert all but the largest maps into open sea. To save the map by itself, answer with the string "m" when "saving the game" (described in more detail below).

The other way to build is by using a text editor and following the format exactly. The terrain data is a number of long lines of characters. The length of the lines must be exactly the width of the map. The characters for terrain types are defined by the period. The coordinate system is Cartesian oblique, with the y axis tilted to form a 60-degree angle with the x axis. Thus, landforms in the mapfile should appear to be leaning to the left, if they are to appear upright during play.

Maps should have some distinguishing terrain on the northern and southern edges; also, remember that the default map shape wraps around in the east-west direction, so landforms should match up. If the wraparound is undesirable, a vertical stripe of some otherwise-unused type (such as outer space) is useful to block movement.

Run-length encoding is also available. It is flagged by a numeric digit, followed by any number of digits, followed by a terrain character. The terrain character will be replicated the number of times specified by the digit string. The encoding may be freely intermixed with normal terrain, but cannot extend over line

boundaries.

GLOBALS

Globals include a number of values affecting an entire game. The header is in the following form:

Globals *gametime endtime setproduct leavemap numconds extension*

where *gametime* is the elapsed turns, *endtime* is the last turn of the game, *setproduct* controls whether unit construction is changeable, *leavemap* allows units to leave by the map edges, and *extension* is reserved for extensions (is normally **0**).

numconds specifies the number of winning/losing conditions. Conditions are evaluated at the end of each turn (along with other ways to lose, such as losing all of one's units). A win condition results in the side achieving it winning while all others lose. A losing condition knocks the side out of the game, and the other sides continue normally. Multiple win and lose conditions act as disjunctions - when one is satisfied, something will happen. There is a limit of about 10 conditions or so. All conditions include a starting and ending time for when they are in effect, as well as the number of a side to which it applies, or **-1** it is to be applied to all sides equally.

The first line of a condition is the same for all types:

win/lose type start end side(s)

where *win/lose* is **1** if the condition is for winning and **0** for losing, *start* and *end* are the starting and ending turns for testing the condition, *side(s)* is the number of a side to which the condition applies (or **-1** if it applies to any side), and *type* is the type of condition:

0 Territory; next line is one number, the amount of territory (above if to win, below if to lose).

1 Number of units; next line is list of numbers, one for each unit type. To win, must possess at least that many of each type. To lose, must be at or below that many for *all* unit types simultaneously (the default losing condition is this one, with all zeros). To make this condition apply to only one unit type, set all the other numbers to very low (for win condition) or very high (for lose condition) values, outside the range normally occurring during a game.

2 Quantity of resources; next line is a list of numbers, one for each resource type. Winning and losing is same as for unit types.

3 Possession; next line is three numbers, the first two representing the coordinates of a hex, and the third the number of a unit type (or **-1** to indicate all unit types). To win, must have a unit of the specified type on the hex. To lose, do not have a unit on that hex.

Each condition has a sort of dual identity, since it is interpreted slightly differently, depending on whether it is flagged for winning or losing. The logic is a little twisted perhaps - special conditions should have all cases tested carefully before a scenario is released.

SIDES

Sides need relatively little information stored about them, particularly for a scenario. The header is simple:

Sides *numsides detail extension*

where *numsides* is the number of sides recorded, and *detail* is the level of detail that was recorded.

The side's name can be set using the naming or **C**all command, when the cursor is not over a unit, or if an argument is supplied (this will be the number of a side, and can be used in build mode to change the names of other sides).

The exact contents of each level of detail are as follows:

1 Name of side only is saved. This is sufficient for many scenarios.

2 Name, attributes, and political status and production counts (used for numbering unnamed units) saved.

3 All of above, plus view of explored part of world.

4 All of above, plus hosts and player types, plus statistics if used.

9 All possible data. This is the level used for saved games. (At the moment, this is equivalent to level 4, but may include more data in the future.)

The data for each level begins on a new line, and some levels need several lines (such as the view data). The best way to study the layout is to examine a saved game.

UNITS

Units have quite a few attributes, nearly all of which must be saved. The header is as simple as for sides:

Units *numunits detail extension*

where *numunits* is the number of units saved, *detail* is the level of detail that was recorded, and of course *extension* is the usual extension with only a value of **0**. There are several levels of detail:

1 Type, name, and position, and side. The re-created units will be fully supplied and awake. Neutral units have a side of -1. Units capable of construction will be idled, so there should be at least one movable unit present.

2 All of above, plus scalar attributes and supply amounts, all on one line. (Look at a saved game or the source code to interpret the numbers.)

3 All of above, plus orders and standing orders if defined.

9 All possible data. This is the level used for saved games. (Same as level 3 at present.)

Build mode offers a number of way to manipulate units on any side, as described in the next section.

SCENARIOS

The **-B** command line option starts up *xconq* in *build mode*, where many additional actions are possible, all oriented towards the editing of game state. All commands can be performed on any side's units, and none of the machine players will move (this can be toggled by using the quit command 'Q', then saying 'n' to the confirmation question). Build mode allows some other acts:

Move any unit anywhere instantly, using the moveto command 'm'.

Create any unit anywhere, using '\' (prompts for unit type, argument specifies side, defaults to neutral).

Modify terrain anywhere, as described under the map section.

See all units everywhere, using 'V'

Do anything to any unit as if it were your own, using the usual commands.

Once all desired changes have been made, you may wish to allow machine players to move a little, just to randomize things a bit. Then use the normal game save command. In build mode, you will be asked to enter a string indicating sections and levels of detail. The string should contain a character for each section you want written out - 'g' for globals, 'u' for units, and so forth. The letters 'u' and 's' must each be followed by a digit indicating the level of detail desired, as described earlier. The default string is **"ms1u1"**, which will write a mapfile with the map, side names, and unit types/names/positions/sides, which is sufficient for many interesting scenarios. The file that will be written is always called **"random.scn"**, is written into the current directory, and should be renamed as desired. After sufficient playtesting, it may be added to the scenario menu, just by adding its name to the file **"mapfiles"** in the library directory.

Scenario construction is not for everybody. Since the processes are semi-internal, the error-checking is not as extensive. For instance, you can load a submarine with battleships as passengers. There are also more subtle questions of balance, which are usually revealed by repeated play of the scenario. As a rule, the lower levels of detail are safer to use - level 1 details for sides and units are often simple enough to be typed in or edited by hand.

EXAMPLES

A period description with only infantry and cities:

```
Xconq 0 -+---- Very simple test period
Period 0

"generic" period-name

"standard" font

"i" "infantry" "moves around" utype
"@" "city" "makes infantry units" utype

"+" "land" "green" ttype  ; must always have at least one terrain type

1 @ in-country
100 land @ favored

@ first-unit
i first-product

1 i @ make
true @ maker
1 i @ capacity  ; makers need to be able to hold or be held by products

1 i speed

0 t* i moves

begin{notes}
This is just a test.
Kids, don't try this at home!
end{notes}

end
```

A tiny map with two cities, including an "empire" period description to ensure meaningful city definitions:

```
Xconq 1 --+--+ Tiny map;
just an example...
empire.per
Map 9 5 100 1 0
:.....
:.+++++..
:..+^^^..
:...+^^..
:.....
Units 2 1 0
@ New Cork 2,3 -1
@ Old York 4,1 -1
```

LOSSAGES

Several words are marked NIY, meaning that although they can be used, the code does not actually take them into account.