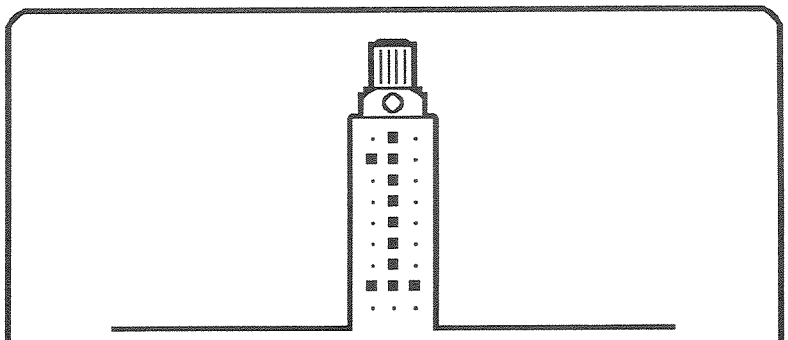
THE GENESIS DATABASE SYSTEM COMPILER: USER MANUAL

Don Batory

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188

TR-90-27 August 1990



Department of Computer Sciences
The University of Texas at Austin

The Genesis Database System Compiler: User Manual

Don Batory
Department of Computer Sciences
The University of Texas
Austin, Texas 78712

August 1990

Features and Known Bugs

DaTE. Not everything is perfect. DaTE has bugs that trash memory or make illegal memory references. Unfortunately, some of these bugs are very difficult to track down and fix, due to the memory management scheme of the MacIntosh. Simply put, the errors aren't always reproducable. You'll discover the do's and don'ts on your own; it isn't difficult to make DaTE work for you. At least with a bit of alchemy. I highly recommend that you install a version of **MacsBug** to trap these errors.

DaTE tries to check for design errors. Most of the time, it catches mistakes and reports them to you. However, if you create a storage system which references itself, don't expect things to work.

We've already alerted you to DaTE's other quirk - the need for the Convert Refs application. See the end of Phase 1 for an explanation.

The size (in terms of number of lines) of each layer, as registered in DaTE is not accurate. We are in the process of determining how to count line numbers (should comments be included, etc.) and will be updating these numbers shortly. Yes, there are 70,000+ lines of Genesis source, but removing headers and comments (which are essential to understand what's going on) will reduce this number.

Finally, don't mess with trying to define new layers or altering existing layer definitions, unless you know what you are doing. Technical reports exist which explain the basic ideas. Once you know what each button, etc. means, then you can proceed to experiment.

Genesis Source. You may have noticed that there is source code or references to a db_cache recovery, grid file structure, and references to Ziv-Lempel encoding. Right now, db_cache and grid have some bugs which we are trying to track down. Ziv-Lempel encoding does work on Sun3s, but does not work on Mac IIs. A port revealed that the Ziv-Lempel algorithms we are using require enormous byte arrays, larger than Think C can handle. So we eliminated it.

Also, B+ trees are known not to work when at most two records fit per node/block. Debugging B+ tree code is painful, and this is still on our to do list.

Gdefine & Gdml. Again, nothing is perfect. Gdml somehow leaks memory. A consequence of this is that very long scripts will not execute to completion before Gdml blows up. You can go quite some time before something happens, but something will happen eventually.

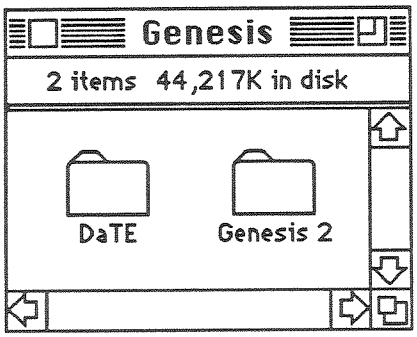
Another problem is that presently, all genesis databases are stuffed into individual volumes of a single, predefined size. For Mac IIs, the size is 2000 blocks of 512 bytes. To change this, you'll need to edit genesis_tun.h, which is subfolder **headers** in the **GenSource** folder. As a general rule, and header file (genesis, jupiter, or system) _tun.h contain constants that you can change. Just be careful.

Introduction

Genesis is the first software building-blocks technology for database management systems. It is also one of the first examples of large scale software reuse. This paper describes how to assemble DBMSs using Genesis. The design and implementation techniques utilized in Genesis are described elsewhere. Instructions on how to install Genesis on a Mac II are given in the Appendix along with a list of known bugs.

Getting Started

Genesis consists of a configuration editor (DaTE) and prewritten software modules called **layers**. On Macintoshes, the **Genesis** folder contains all Genesis software. The **DaTE** folder contains DaTE and its files; the **Genesis 2** folder contains building-block source code:

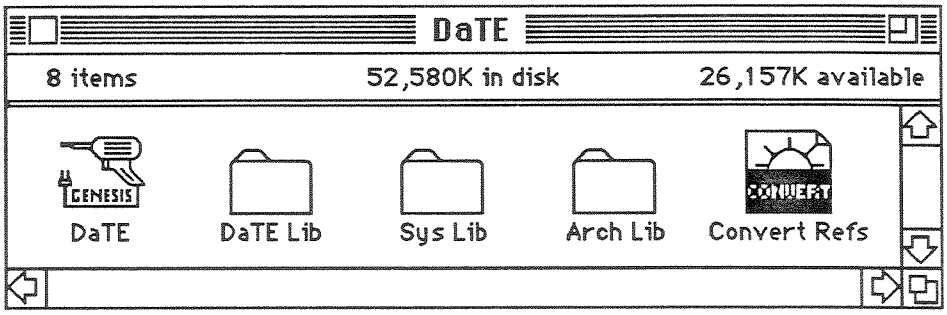


There are three phases in the life-cycle of a Genesis DBMS: specification, assembly, and usage. The **specification phase** involves using DaTE to define a target database management system. The output of DaTE is a set of **configuration files** that specify the interconnections between layers of the target system. The **assembly phase** is the actual creation of DBMS executables. This is accomplished by compiling the generated configuration files with the Genesis library. The **usage phase** deals with assembly validation, schema creation, compilation, database loading, and database processing.

The following chapters will explain each of these phases in more detail. As a running example, we will show how an approximation to University Ingres can be generated.

Phase I: DBMS Specification

The most complicated and most interesting phase of DBMS generation is that of specification. **DaTE** (or Database system Type Editor) is a graphical language for composing software building-blocks. The DaTE folder contains the DaTE and Convert Refs applications and three folders: DaTE Lib, Arch Lib, and Sys Lib. **DaTE Lib** has definitions of all primitive Genesis building blocks. **Arch Lib** and **Sys Lib** respectively contain the architectures and systems generated by DaTE.

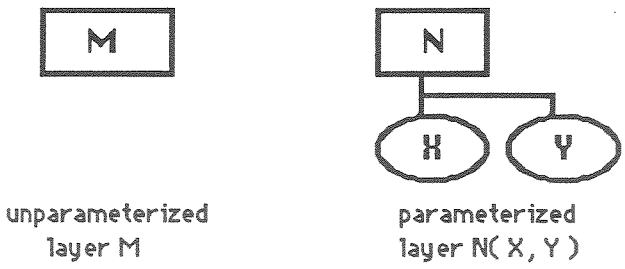


Clicking DaTE begins its execution, which starts by reading primitives from the DaTE library. **Convert Refs** is needed only to transport architecture designs from one disk to the next. We'll consider its use at the end of this Phase.

> (Note: as of this writing, the **Sys Lib** and **Arch Lib** folders are a convention which we strongly recommend readers to follow; they are not required by DaTE. The **DaTE Lib** folder is required).
>
> Historical note: Primitives in the DaTE/Genesis library were considered, at one time or another, to be software ICs. The Genesis icon - a wire wrapper or soldering iron - was chosen to symbolize the interconnection of ICs into software systems. The permanence of this icon remains to be seen.
>
> The name DaTE - Database system Type Editor - was chosen at a time when the distinction between layers and parameterized types was not recognized. The permanence of "DaTE" as a name also remains to be seen.

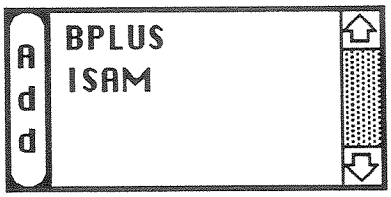
DaTE Diagrams

Building-blocks of Genesis are parameterized **layers**, a concept akin to a parameterized type. DaTE depicts layers as boxes and parameters as ovals. Layers M and N are shown below; M is unparameterized and N has two parameters, X and Y:



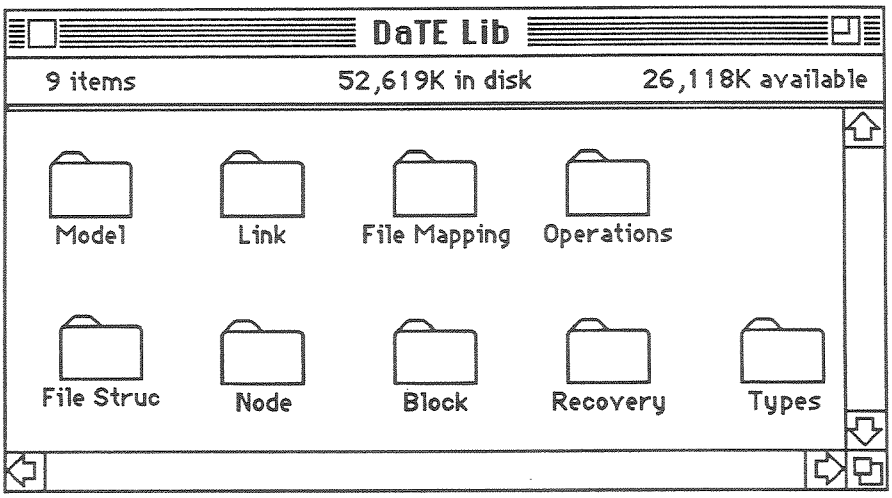
An **architecture** is a rooted graph. DaTE supports the definition of four progressively more complex types of architectures: file structure, storage system, network, and relational. An architecture is **complete** if it has no unbound parameters (i.e., no ovals). Complete architectures are treated as primitive, unparameterized layers by DaTE.

A fundamental concept in DaTE is the **software bus**. It is an abstract construct that allows multiple layers to occupy the same position in an architecture. Software busses are depicted as a scrollable window. The bus below lists the layers (or architectures) BPLUS and ISAM:



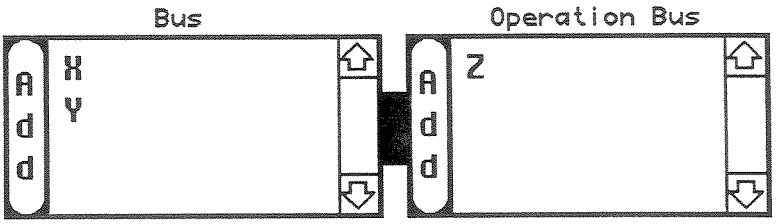
The **Add** button admits new entries to a bus. An entry is deleted by clicking it and choosing the **Remove** option from the displayed popup menu.

Layers and architectures belong to classes. There are the classes of file mapping layers, link layers, etc., as well as the class of file structure architectures, the class of storage system architectures, and so on. If you open the DaTE Lib folder, you'll see the classes presently available in Genesis. **File Struc**, for example, is a folder containing different file structure layers. **Recovery** is a folder containing different page-based recovery layers.



Getting back to software busses, DaTE restricts entries of a software bus to belong to a single class. Thus, all entries of a bus are storage systems, or all are link layers, etc. In principle, it may be possible (and useful) to have polymorphic software busses; this is a concept that needs to be explored in the future.

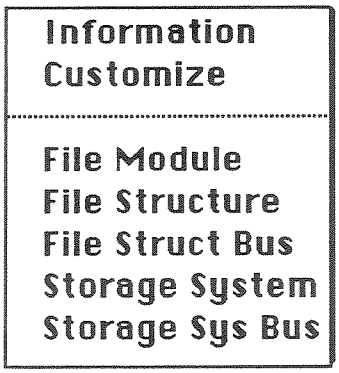
A bus can be extended by an **operation bus**, which permits the entries of a bus to reference special operations. An operation bus is attached to the software bus it is to complement:



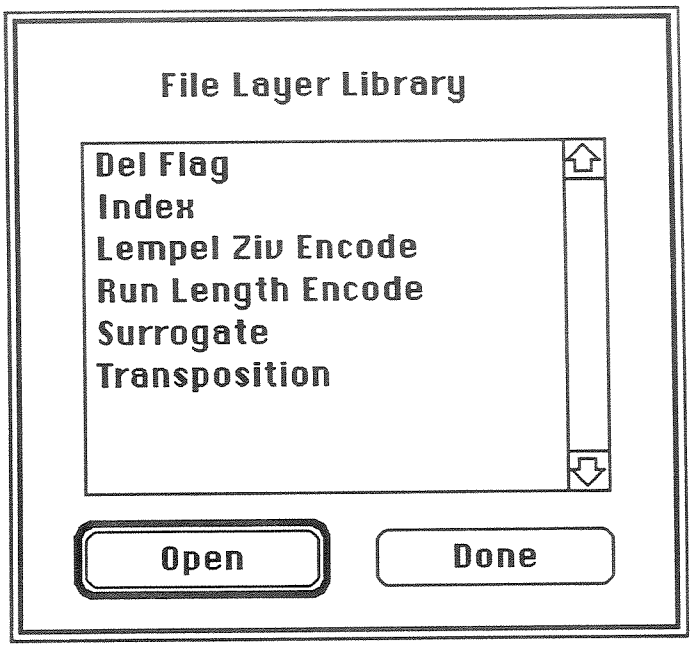
The semantics are straightforward: any entry in the primary bus may reference any entry in the operation bus. The above figure shows entries X and Y in the primary bus, and they may reference entry Z of the operation bus. Operation busses arise only in FMS and DBMS specifications.

Instantiating Parameters and General Editing Rules

Parameter instantiation in DaTE is accomplished by clicking an oval. The standard response of DaTE is to display a menu, like the one below:



Selecting **Information** displays a help window on the selected item. **Customize** lists the customizable options of a layer, and allows options to be enabled or disabled. Selecting an entry below the dotted line causes a scrollable **library window** to be displayed. The members of the library are legal layers or architectures that can instantiate the selected parameter. A library window for File Layers is shown on the top of the next page.



Occasionally, parameters are bound incorrectly (or better choices are later discovered). Rebinding a parameter is accomplished by clicking the box that is presently bound to the parameter. Choosing an alternative binding causes the previous selection to be overridden. Because DaTE imposes a top-down design methodology, all hierarchical bindings of the earlier layer may need to be erased as they might no longer apply to the new module. DaTE tries to save such bindings whenever possible.

Architectures and Systems

As mentioned earlier, an architecture is a rooted graph of primitive layers. There are file structure, storage system, network, and relational architectures. A **System** is a composition of one or more architectures and supporting primitive layers, such as data types, recovery, and special operations. File Management Systems (FMSs) and Database Management Systems (DBMSs) can be defined by DaTE. Configuration files, which are used in Phase 2 to assemble target systems, can be generated for FMSs and DBMSs.

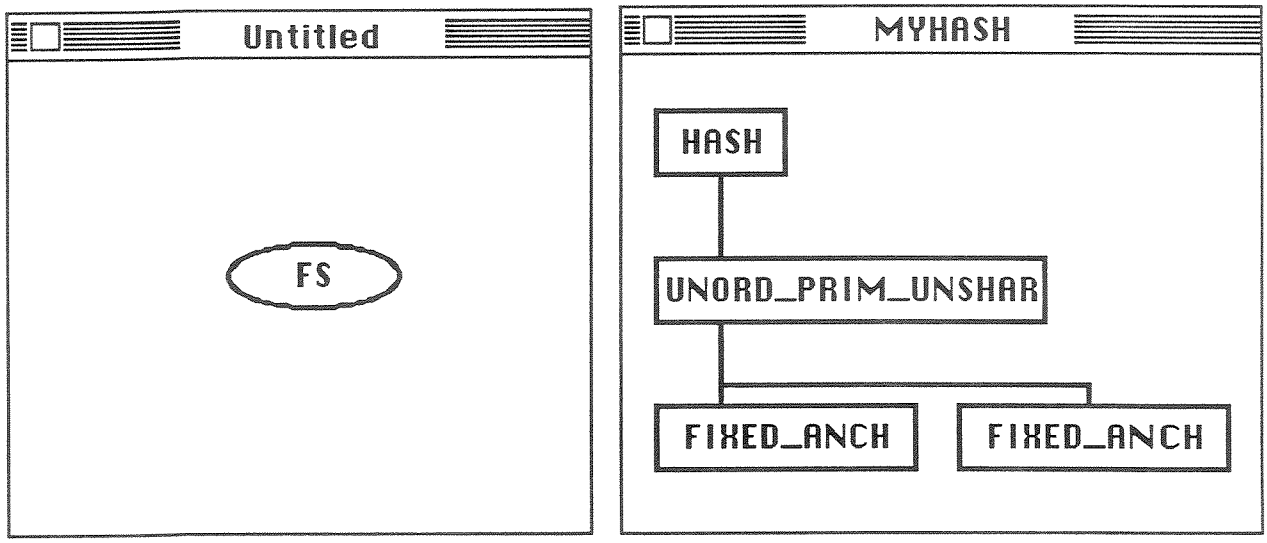
In the following sections, we explain how architectures and system are specified, starting with simplest and progressing to the most complicated.

> Note: Each architecture and each system is stored by DaTE in its own file. Architectures, systems, and configuration files quickly become numerous, and placing them in a single folder is not a good idea. We strongly recommend that architectures be stored in the **Arch Lib** folder and systems and their configuration files be stored in the **Sys Lib** folder provided with DaTE.

Creating a File Structure Architecture

A file structure is a composition of layers that provide the most primitive file storage and retrieval capabilities needed for DBMS operation. Genesis decomposes file structures into three distinct layers: FS (file storage), logical block (or nodes), and physical block.

A file structure is created in DaTE by pulling down the **File** menu, selecting **New**, and then **File Structure**. An empty window is then displayed (a):



(a) (b)

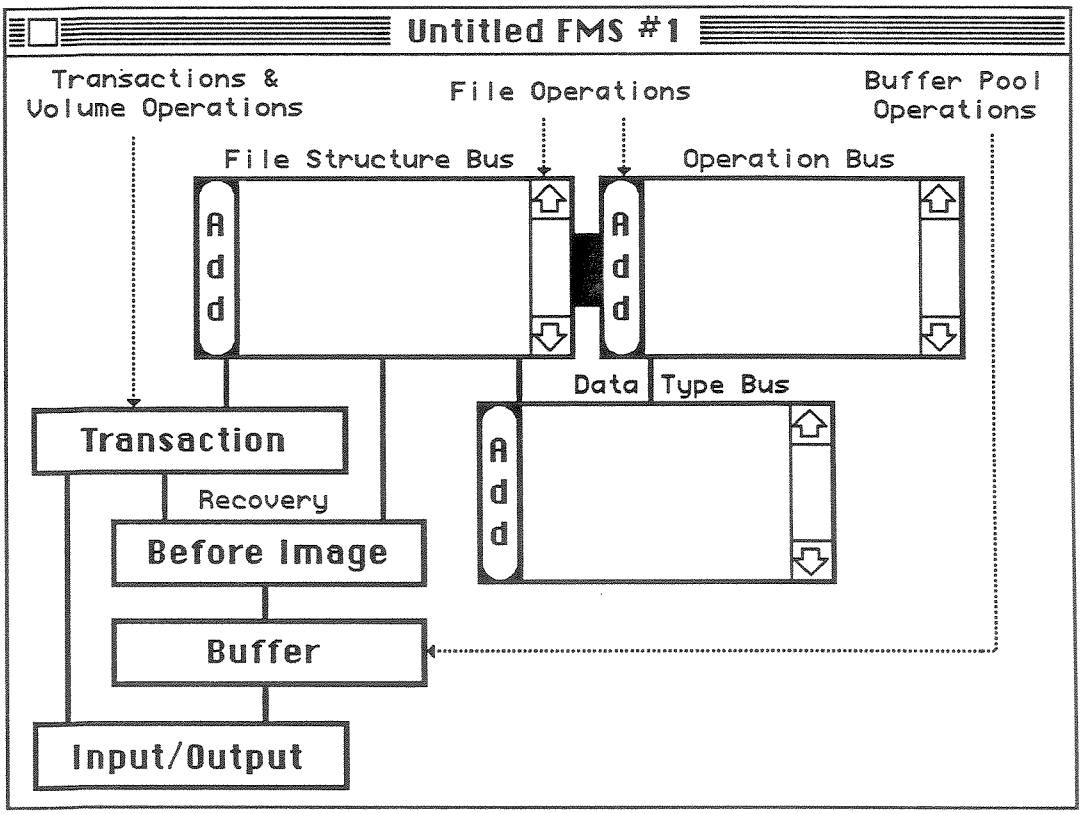
The window contains a single FS oval, indicating that a file storage layer must be specified. Clicking the oval causes a library window to appear that lists all FS layers known to Genesis. Clicking a layer selects it, and its box is displayed in the window. Depending on the layer, one or more Node ovals will hang from the layer box. Clicking these ovals in the same way allows Node implementations to be selected. Nodes have Blocks as parameters, thereby producing a tree of three levels, as shown above in (b). This particular window shows a HASH file structure whose data nodes are implemented by the UNORD_PRIM_UNSHAR layer (i.e., unordered records stored in a primary block with unshared overflow), and primary and overflow physical blocks are implemented by the FIXED_ANCH layer (i.e., fixed-length records with anchored physical addresses).

For a file structure architecture to be referenced later, it must be named and saved. This is accomplished by pulling down the **File** menu and selecting **Save** or **Save As...**. The above architecture was saved with the name 'MYHASH'.

Creating a FMS

A **file management system (FMS)** is the kernel of a DBMS. It provides basic access methods, buffer management, and recovery capabilities necessary for DBMS operation. Genesis provides a standardized architecture for FMSs, where the design customization decisions have been factored into the selection of file structures, special operations, data types, and a recovery layer.

An FMS is created in DaTE by pulling down the **File** menu, selecting **New**, and then **FMS**. An empty FMS window is then displayed:

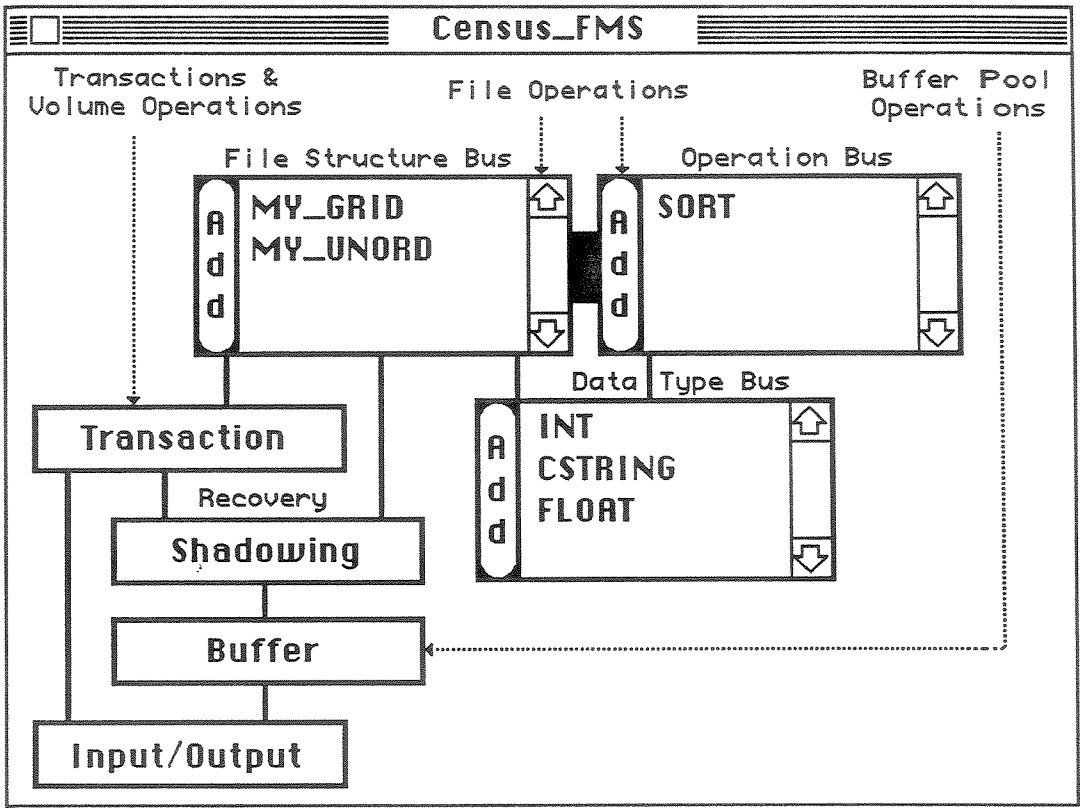


The above mentioned design decisions are entered in three busses and a field. Each labeled in small font as a prompt to an FMS designer. **Before Image** (Page) logging is assumed as the default implementation of recovery.

> Note: **Transaction**, **Buffer**, and **Input/Output** classes in an FMS architecture are not customizable in this version of DaTE, as Genesis provides only a single implementation (i.e., layer) for each. When multiple layers are available, they too will be customizable.

Also note that an FMS window documents the routing (via dotted lines) of user-issued operations. Volume and transaction operations are serviced by the Transaction layer; buffer pool operations are handled by the Buffer layer. File operations are processed by either file structures or special operations. Observe that an FMS is not a strict hierarchy of layers, where all operations are transformed by layers in a top-down manner.

A possible FMS for a census database is shown below. It provides MY_GRID and MY_UNORD as primitive file structures. The SORT operation is included, along with the data types INT, CSTRING, and FLOAT. Shadowing is used for volume recovery.



To generate the configuration file of an FMS, pull down the **File** menu and select **generate.**

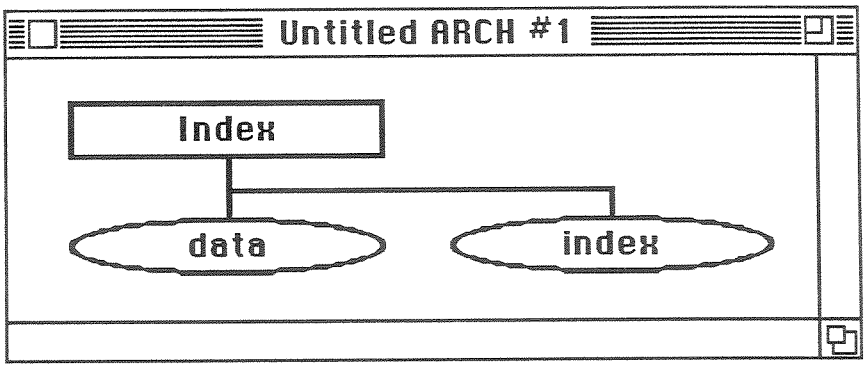
Note: Again, we recommend that FMSs and their configuration files be placed in the **Sys Lib** folder, so that they are separated from their architecture components. The files that are generated are text files that contain C precompiler directives and macro definitions. As we'll see in a later section, these are the files that are compiled with Genesis source to produce DBMS executables.

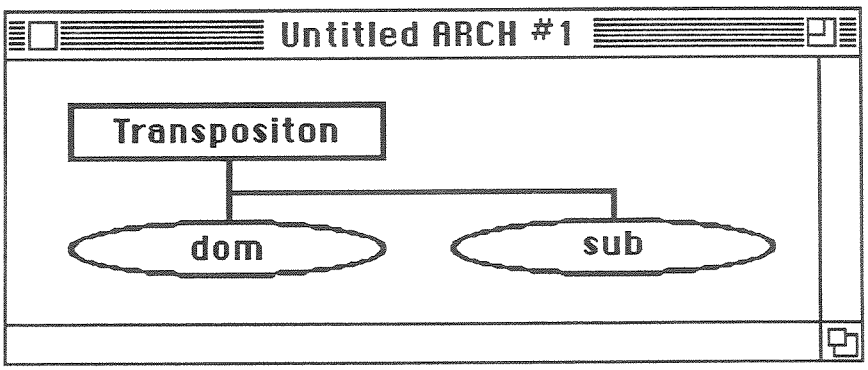
Note: As a general rule, one probably doesn't want to generate only an FMS. When a DBMS is generated, its underlying FMS is also generated. In fact, generating an FMS produces a _fms.h file, while generating a DBMS produces both a _fms.h file and a g_dbms.h file. For now, it is instructive to see what is going on internally with DaTE, although the generation of an FMS is likely to be a rare event for most Genesis users.

Creating a Storage System Architecture

A **file mapping layer** maps an abstract file to one or more concrete files. Examples include mapping a file to an inverted file (i.e., indexing), and mapping an uncompressed file to a compressed file (i.e., compression). Parameters of a file mapping layer are implementations of the concrete files that it generates. A **storage system architecture** is a composition of file mapping layers that terminate with file structure architectures.

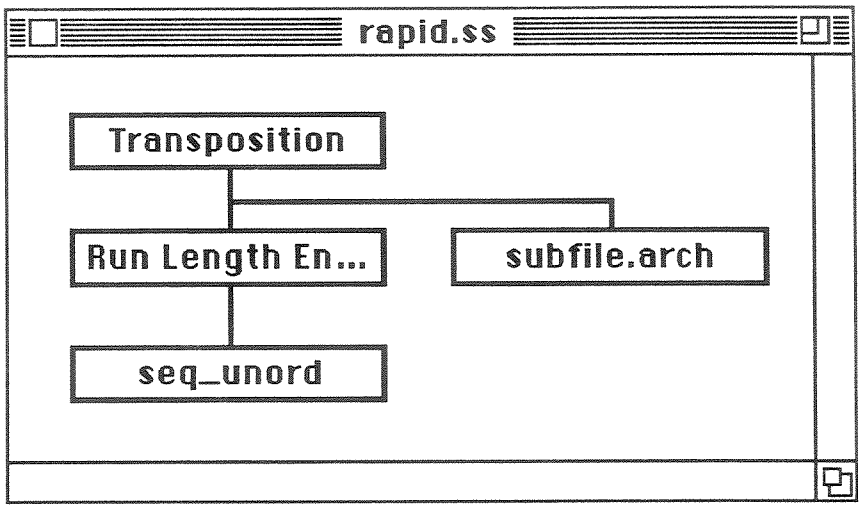
A storage system is created in DaTE by pulling down the **File** menu, selecting **New**, and then **Storage System.** An empty storage system window is then displayed. As examples of file mapping modules, the following two windows show the selection of indexing and transposition layers. **Index** maps an abstract file to a data file and zero or more index files. The data file implementation is specified by parameter **data** and the index file implementation by parameter **index.** Similarly, **Transposition** maps a file to a series of concrete subfiles, one dominant subfile and zero or more subordinate subfiles. Their implementations are specified via parameters **dom** and **sub.**

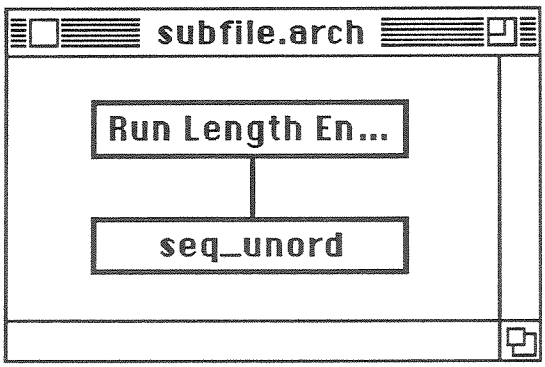




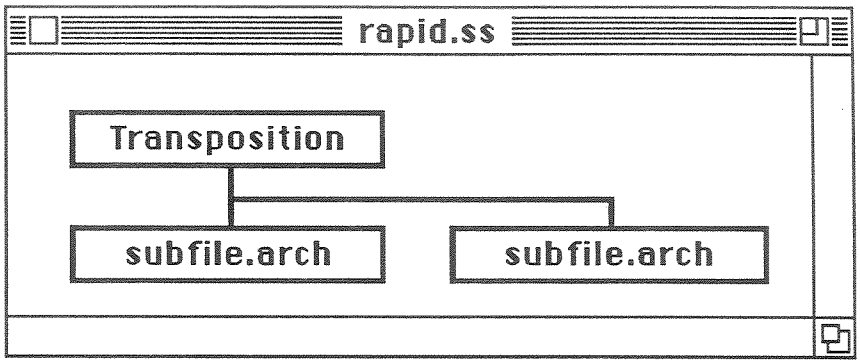
Note: The concept of dominance is fundamental to conceptual-to-internal mappings. Basically the idea is that a conceptual file is mapped by DBMS software to multiple internal files. One of the internal files is distinguishable as its records are in 1-to-1 correspondence with conceptual tuples. This is the **dominant** file. Other internal files, called **subordinate**, do not have this property.

Consider the storage system used by Rapid, a statistical DBMS. Rapid mapped schema-defined files to transposed files, where each column was run-length compressed before being stored in a sequential-unordered file structure. This storage system is defined in two storage system windows: **rapid.ss** and **subfile.arch.** Rapid.ss maps a schema-defined file to its dominant internal counterpart. Subfile.arch maps subordinate subfiles to their internal counterparts.



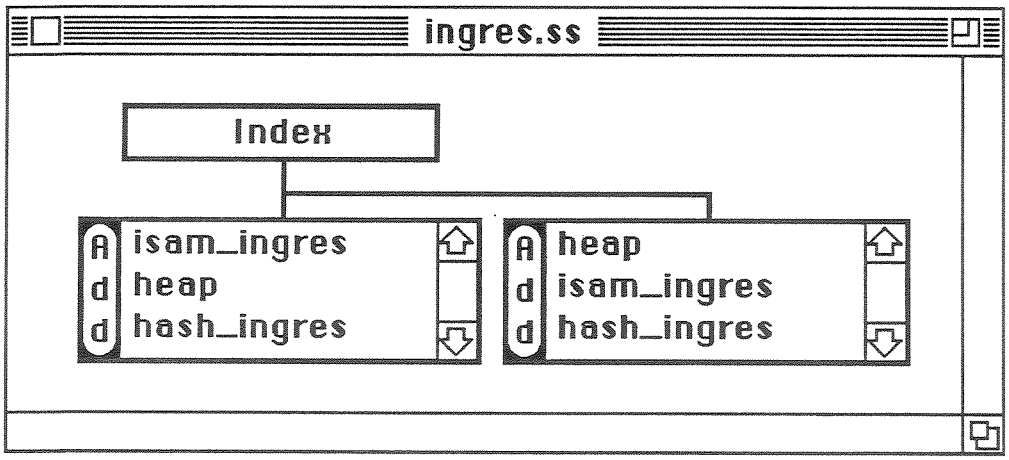


While it seems odd not to have Transposition call subfile.arch twice (as implementations of both dominant and subordinate files are the same), DaTE permits only one architecture reference per dominant mapping. (It turns out that permitting multiple storage system references significantly increases DaTE's complexity without providing greater expressibility. Otherwise, there is no apriori reason why it cannot be handled).



A composition that cannot be defined directly in DaTE....

As another example, consider the storage system of Ingres: it maps schema-defined files to inverted files, where data files and index files can be selectively implemented by hash, heap, or isam structures. The multiplicity of implementation choices is captured by a pair of file structure busses.



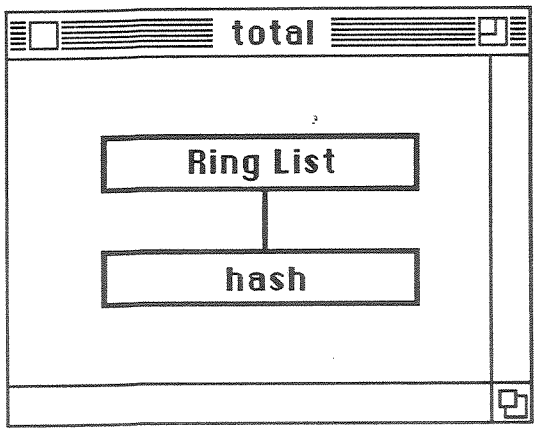
Note: The last entry on a software bus is the default mapping. Thus, data files in ingres.ss default to hash-based structures if no storage structure directive is provided. (These directives are specified in schemas, which is discussed in Phase 3). No other significance is attributed to the ordering of entries on a bus.

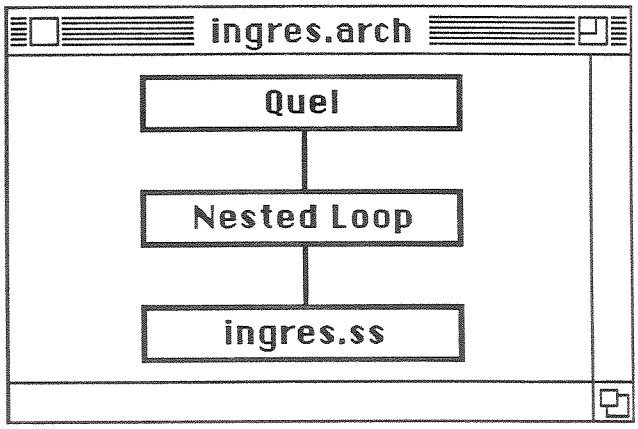
Creating a Network and Relational Architecture

A **network architecture** is rooted by a link layer (or link bus). This layer (or bus) specifies how links - i.e., relationships between files - are to be implemented. The sole parameter of a link layer (or bus) is the implementation of the referenced files, which may be expressed as a file structure or storage system architecture.

A **relational architecture** is rooted by a data model layer (or data model bus). Such layers map nonprocedural data model/data language interface to a procedural network database interface. The sole parameter of a data model layer is the implementation of the links of the network database.

The windows at the top of the next page show a network architecture used in the Total DBMS (i.e., no high-level data model; links are implemented by ring-lists,and files are stored in hash-based structures), and the relational architecture of Ingres (i.e., QUEL as the data model/language, nested loop implementations of links, and files stored in the Ingres storage system).



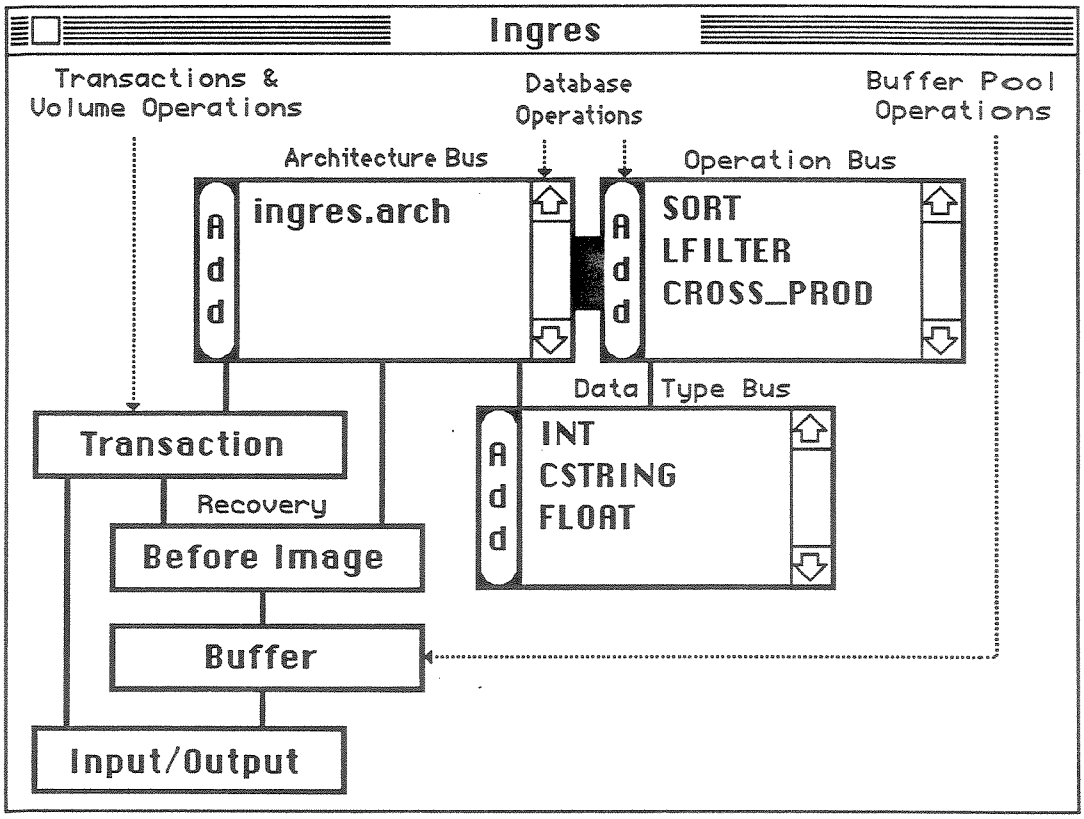


Creating a DBMS

A DBMS is defined in a manner identical to that of FMSs: architectures, special operations, data types, and a recovery layer must be specified. The only significant difference is that relational, network, and storage system architectures are referenced instead of file structure architectures.

A DBMS window for our approximation of Ingres is shown below: the architecture is Ingres.arch; special operations are SORT, LFILTER, and CROSS_PROD; data types are INT, CSTRING, and FLOAT; and recovery is handled by Before Image logging.

Note: LFILTER is a layer required for processing cyclic queries. For further details, see the DaTE help menu.



Statistics about the size of the generated DBMS can be obtained by pulling down the **Misc** menu and selecting **Statistics**. A window similar to the one shown at the top of the next page will be displayed:

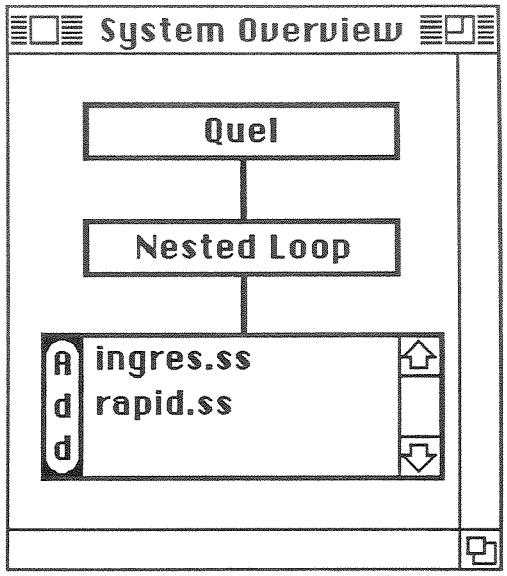
Database Management System Statistics

	Lines of Code		
Manager/Layer	Total	Selected	% of Library
System Managers	6435	6435	100%
System Layers	4863	4863	100%
System Utilities	1427	1427	100%
File Manager	42463	21485	50%
File Layers	6581	1389	21%
Link Layers	7456	982	13%
Model Layers	2468	1234	50%
Totals	71693	37815	52%

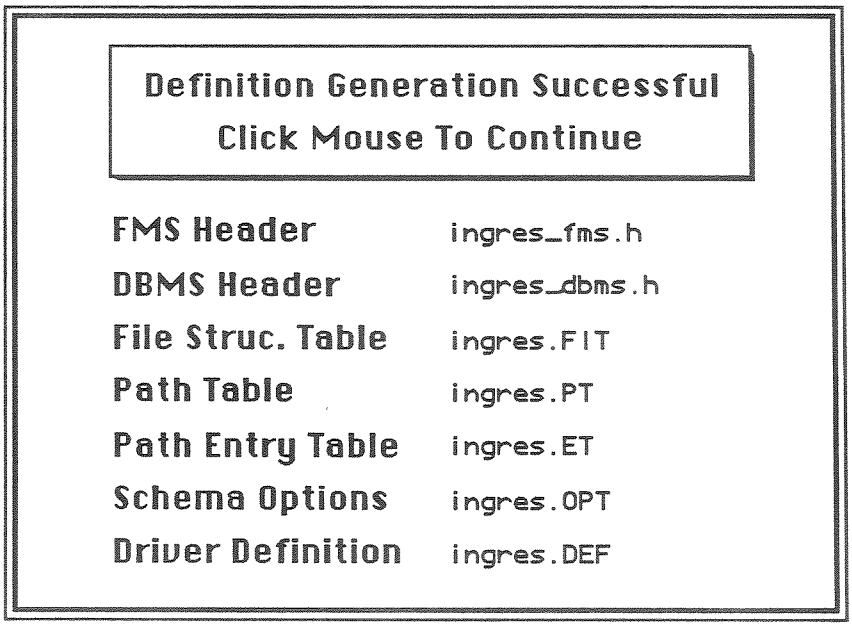
Of the 71K+ lines of code in the Genesis libraries, approximately 52% is referenced in the Ingres DBMS. (Lines that are unreferenced are not included when the Ingres DBMS is assembled). A similar window exists for FMSs.

> Note: **System Managers** and **System Utilities** refer to a standard package of ADTs (queries, into-lists, etc). that are referenced by virtually all DBMS layers. **System Layers** is a generic name given to layers listed on the DBMS's operation bus. **File Manager** refers to FMS code that is generated. **File Layers**, **Link Layers**, and **Model Layers** refer to file mapping, link, and data model layers that are referenced.

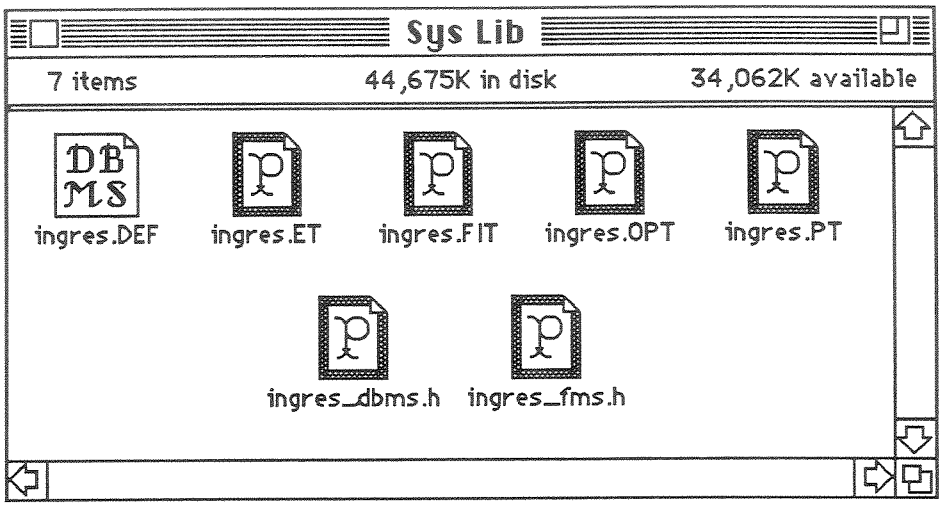
A DBMS typically supports only one architecture. However, if one wants the 'union' of several different architectures (to have the capabilities of several individual DBMSs), one can click multiple architectures onto the Architecture Bus. A composite architecture is formed by taking the union of all data models referenced and placing them on a data model bus, the union of all referenced link layers is placed on a link bus, and the union of all storage systems is placed on a storage system bus. The composite architecture of a DBMS can be viewed by pulling down the **Misc** menu and selecting **DBMS Overview** when the DBMS window is active. The union of Ingres.arch and rapid.ss is show in the figure at the top of the next page.



To generate the configuration files of a DBMS, pull down the **File** menu and select **generate.** The following window will be displayed:



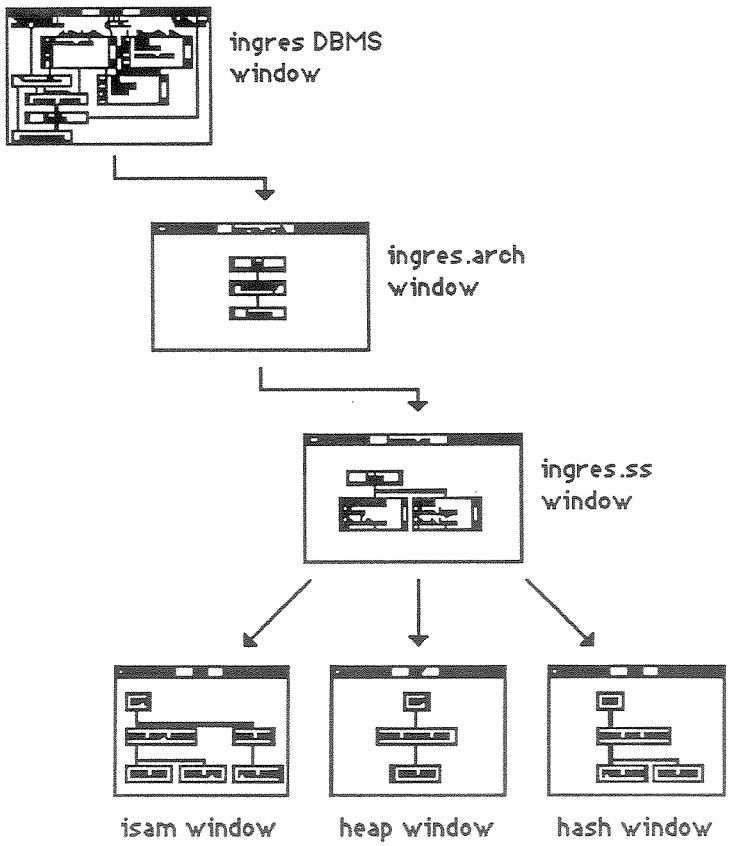
The **FMS Header** and **DBMS Header** files are compiled with the Genesis library to produce Ingres. The **File Structure Table, Path Table, Path Entry Table**, and **Schema Options Table** are read by DBMS executables to process dml operations. The **Driver Definition** is a DaTE readable document that is a copy of the DBMS window that defines ingres. Provided the **Sys Lib** folder is empty, the results of this generation are shown at the top of the next page.



Note: Again, we recommend that DBMS designs and configuration files be placed in the **Sys Lib** of the DaTE folder.

Window Management and Complex Architectures

Architectures quickly become too large to fit on single windows. This is one reason why DaTE has different windows for different architectures and different systems. DaTE provides a convenient mechanism to navigate among interrelated windows/architectures. The figure below shows relationships between different windows and different parts of a DBMS design:



Recall that complete architectures are treated as unparameterized layers. When one clicks on the box of an architecture and selects **Information**, the window for that architecture is opened. Since architectures can be nested, navigational paths may become long. To retrace to the parent window, pull down the **Windows** menu and select **Back**. Thus, one can navigate the tree of windows of the Ingres DBMS through menu selections.

> Note: At present, we recommend that **at least** one relational architecture be included in a DBMS specification. If a relational architecture is not included, the system that is generated has only a programming language interface. There is no convenient driver that is available, currently, to allow users to explore such systems and issue basic database calls without a considerable amount of programming. If a relational architecture is included, then either SQL or QUEL - a high level query language is made available for issuing basic calls.

Convert Refs

When an architecture is created, there are references to primitive layers and possibly other architectures. DaTE saves these references using the MacIntosh equivalent of absolute pathnames. Thus, if you decide to copy you architectures onto another disk, or simply to move them from one folder to another, you've changed the pathnames of your architectures. DaTE will choke when it tries to read such files again. As a partial fix to the problem, the **Convert Refs** utility is used. Drag it into the folder whose architecture files have been moved. Double-click it to start it to execute. If all pathnames can be converted properly, it will report success. If some pathnames can't be converted, the count of the number of errors is reported. Not exactly useful, but when you run DaTE, it will become obvious which architecture files haven't been properly translated.

Phase 2 - Assembly

A Genesis produced DBMS has two executables: **Gdefine** and **Gdml**. **Gdefine** compiles schemas and creates files for loading. **Gdml** loads empty databases and supports database processing via nonprocedural query languages. In the following, we assume a Think C (formerly Lightspeed C) environment for assembling DBMSs. We will also use ingres as a running example.

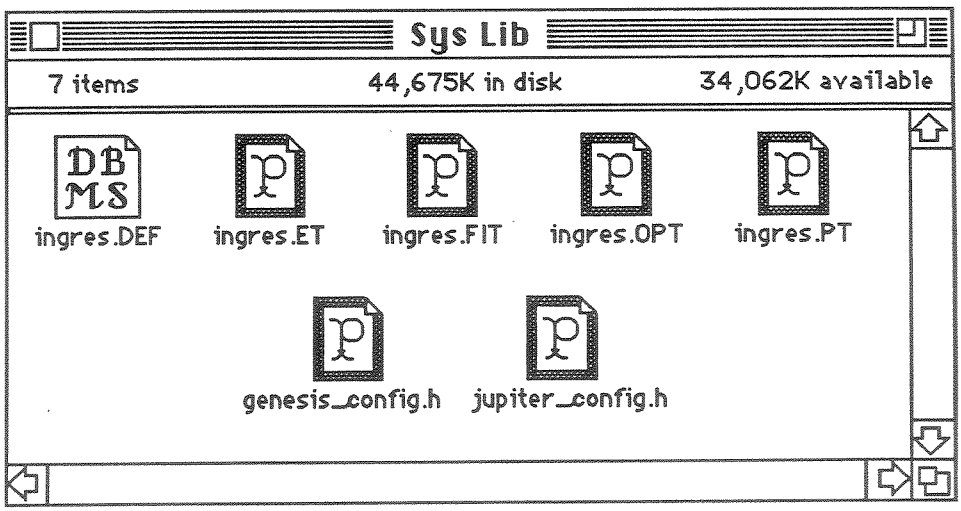
> Note: Ultimately, **Gdefine** and **Gdml** will be merged; it is only for historical reasons that both were developed independently. The compilation of **Gdefine** and **Gdml** takes about 10 minutes. As we'll explore later, Universals can eliminate the need for compilation altogether.

There are two ways to produce executables. Either you build versions of Gdefine or Gdml that containly only the layers that are required for your target system, or you can build a universal Gdefine and Gdml. Universals, as we will call them, are executables that contain virtually every building-block in the Genesis Library. Module interconnections are realized at run-time via dispatch tables. Thus, using Universals, it is possible to go directly from a DaTE specification to DBMS execution, eliminating the need for compilation. Of course, a penalty to be paid for Universals is slightly slower speeds and the inability to implement certain functions. We will explain more about Universals at the end of this chapter.

In the following sections, we'll explain how to compile versions (Universal or otherwise) of Gdefine and Gdml. Producing Gdefine and Gdml executables is a 4-step process.

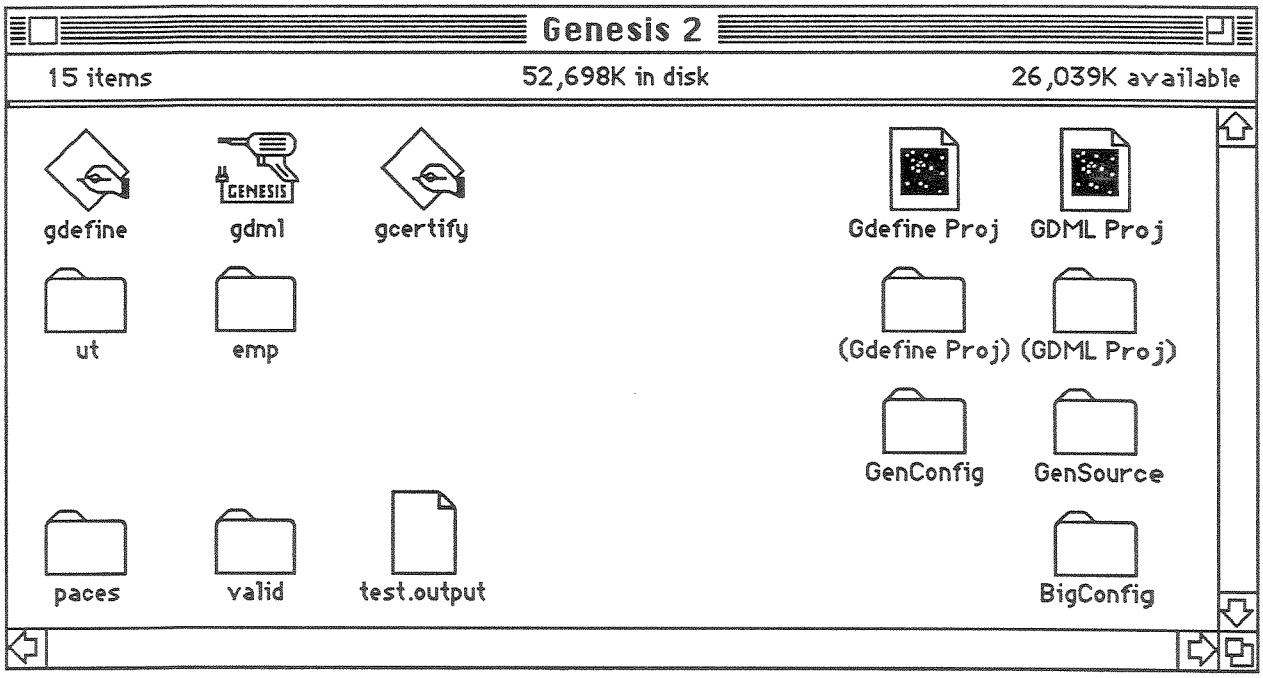
Step 1 - Renaming Configuration Files

Among the header files that are referenced by Genesis source are **genesis_config.h** and **jupiter_config.h**. Both are transcripts of a DBMS design which partially specify the interconnections between Genesis layers. (genesis_config.h deals with dbms layers; jupiter_config.h deals with FMS layers). DaTE distinguishes different config.h files by prepending the name of the DBMS. The first step in compilation is to rename ingres_dbms.h to genesis_config.h and ingres_fms.h to jupiter_config.h. Here what the Sys Lib folder should look like after the renaming:

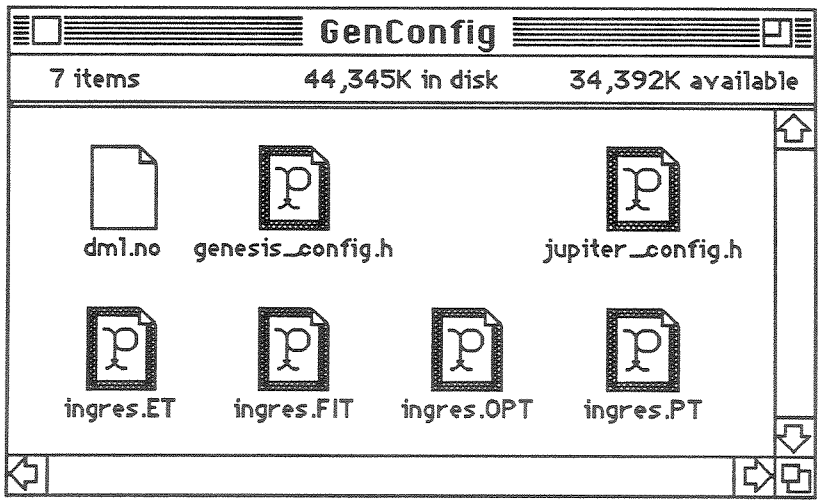


Step 2 - Transferring Configuration Files

The **Genesis 2** folder is shown below. All icons on the right deal with the assembly phase (the Phase we are now explaining); all on the right deal with the usage phase (the Phase dealing with schema compilation and DBMS transaction executions).



Drag all of the ingres configuration files in **Sys Lib** (except for ingres.DEF - it isn't needed) into the **GenConfig** folder. After doing this, open **GenConfig** and you'll see:



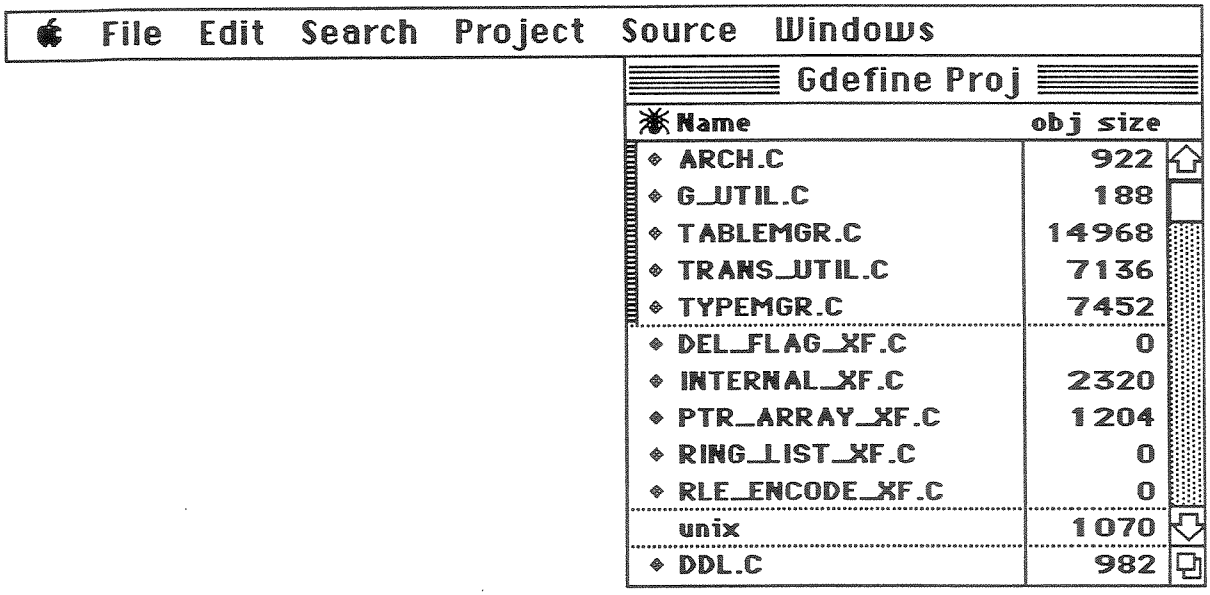
The file **dml.no** is used by **Gdml** to parse dml statements. It should never be deleted or removed from **GenConfig**. (Just in case, there is a backup copy of dml.no in **GenSource** folder, which contains Genesis source).

Hint: **GenConfig** usually contains a **genesis_config.h** and **jupiter_config.h** file. It is easy to misspell these names when renaming. We recommend dragging the renamed **genesis_config.h** and **jupiter_config.h** separately into **GenConfig**; a prompt will ask if you want to delete the old version. (You do). In this way, one is assured that **GenConfig** has the correct files.

Step 3 - Building Gdefine

There are three icons that are relevant to the construction of **Gdefine**:

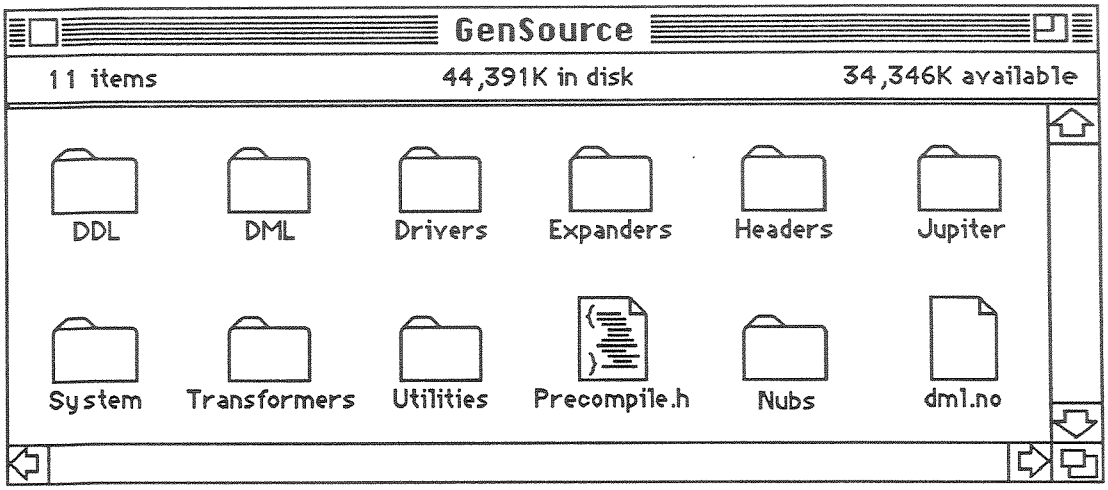
Gdefine Proj is the Think C document that assembles **Gdefine**. **(Gdefine Proj)** is a special folder associated with **Gdefine Proj**. **GenSource** contains Genesis source. Double-click **Gdefine Proj** to open it. You'll be greeted with the following menu bar:



Step 3.1 - Precompiling Headers

A standard set of header files is included in every Genesis source module. As there are almost seven thousand lines of headers, rereading and recompiling them for each module is redundant and time consuming. Think C provides a convenient way to compile headers once. That's what this step is about.

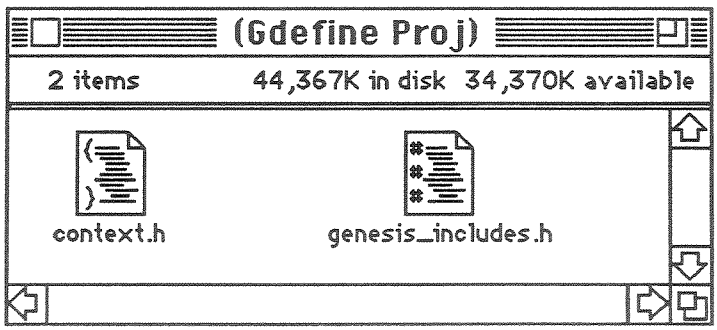
Opening **GenSource** folder you'll see:



Open the file **Precompile.h** by pulling down the Think C **File** menu and selecting **Open**. (The source for **Precompile.h** should be displayed). Now pull down the **Source** menu and select **precompile...** This will initiate the precompilation.

Note: **Precompile.h**, like any of the Genesis source code, should never be moved, updated, or destroyed. In the event that it is unreadable, a backup copy - called **genesis_includes_src.h** - can be found in the **Headers** folder. Also, note the backup copy of **dml.no** in **GenSource.**

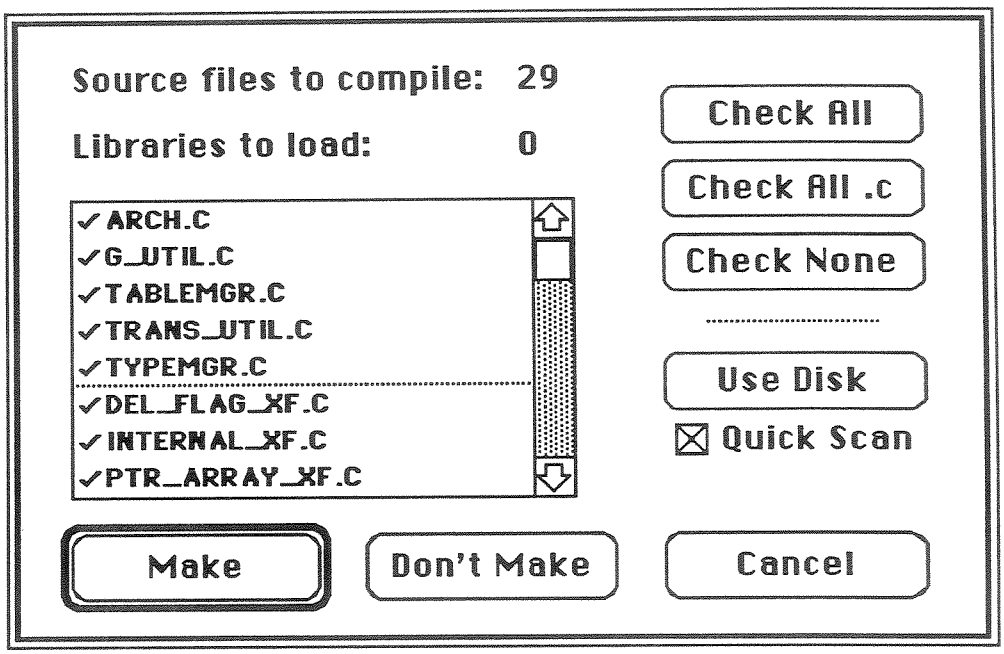
The precompiled header must be stored in the **(Gdefine Proj)** folder under the name **genesis_includes.h**. There will always be a version of **genesis_includes.h** in this folder, so the system will prompt if you want to throw away the old copy. (You do.) The **(Gdefine Proj)** folder should always contain only two files:



Note: The **context.h** file is a small file that tells Think C to precompile Genesis headers for **Gdefine**. If one looks in the **(Gdml Proj)** folder, a similar pair of files exist. The **context.h** file in **(Gdml Proj)** is different, and tells Think C to precompile Genesis headers for **Gdml** in a manner that is different from Gdefine.

Step 3.2 - Producing Object Files

Pull down the Think C **Source** menu and select **Make**. A window similar to that shown below will appear. The number of source files to compile will be listed as 21. This is incorrect. Click the **Use Disk** button to bring this number up to 29. (If you get different numbers, start again from Step 3.1). Now click the **Make** button to begin compilation.



Compilation should take about 4 minutes; the number of lines of code that Think C compiles (some of which generate no object as they deal with layers that are not needed in the target system) is over 70 thousand.

Step 3.3 - Building Gdefine

To link the object modules in Step 3.2, pull down the **Project** menu and select **Build Application.** The executable that is generated should be stored in the **Genesis 2** folder (which will overwrite the previous version of **Gdefine**). You shouldn't have to specify the **Gdefine** name, but you will have to direct Think C's output to the **Genesis 2** folder.

At this time, **Gdefine** is ready to run.

Step 4 - Building Gdml

The procedure for building **Gdml** is identical to that of **Gdefine**, except that the **Gdml Proj** is run and the precompiled header is stored in the **(Gdml Proj)** folder.

When objects are produced, the number of files to compile will be listed as 25. This is incorrect. Click the **Use Disk** button to bring the number to 33. If you get numbers different from these, begin again from the precompilation step.

Compilation should take about 6 minutes; the number of lines of code that Think C compiles (some of which generate no object as they deal with layers that are not needed in the target system) is over 90 thousand.

> Note: The number of actual lines of source in Genesis is about 70 thousand; there is about 7 thousand lines of headers. The number of lines that are reported compiled by Think C involves a certain amount of double counting; some headers aren't precompiled and thus must be read (and compiled) multiple times.

Before Proceeding to Phase 3...

Make sure that the versions of **Gdefine** and **Gdml** that you created exist in the **Genesis 2** folder, and are not in some subfolder. Placing **Gdefine** or **Gdml** in the wrong directory isn't difficult, and doing so without noticing can cause a lot of errors and confusion. You can be assured that Gdefine and Gdml are in the right folders if you are prompted - when Think C is building your application - when you replace the existing Gdefine or Gdml.

Universals

It is possible to build versions of Gdefine and Gdml that contain virtually all layers of Genesis. The configuration files for Universals are found in the **BigConfig** folder of **Genesis 2.** That's the folder in the lower right-hand corner. If you open **BigConfig** up, you'll see the genesis_config.h and jupiter_config.h files and the Universals created from them.

Virtually all layers can be included in Universals, but there are exceptions. Universals can only have one recovery layer. So if you want to differentiate Universals that rely on shadowing from those that rely on before-image page logging, you'll have to create multiple versions of the Universals.

Open the jupiter_config.h file and scroll to the bottom. That's where you'll see constants like:

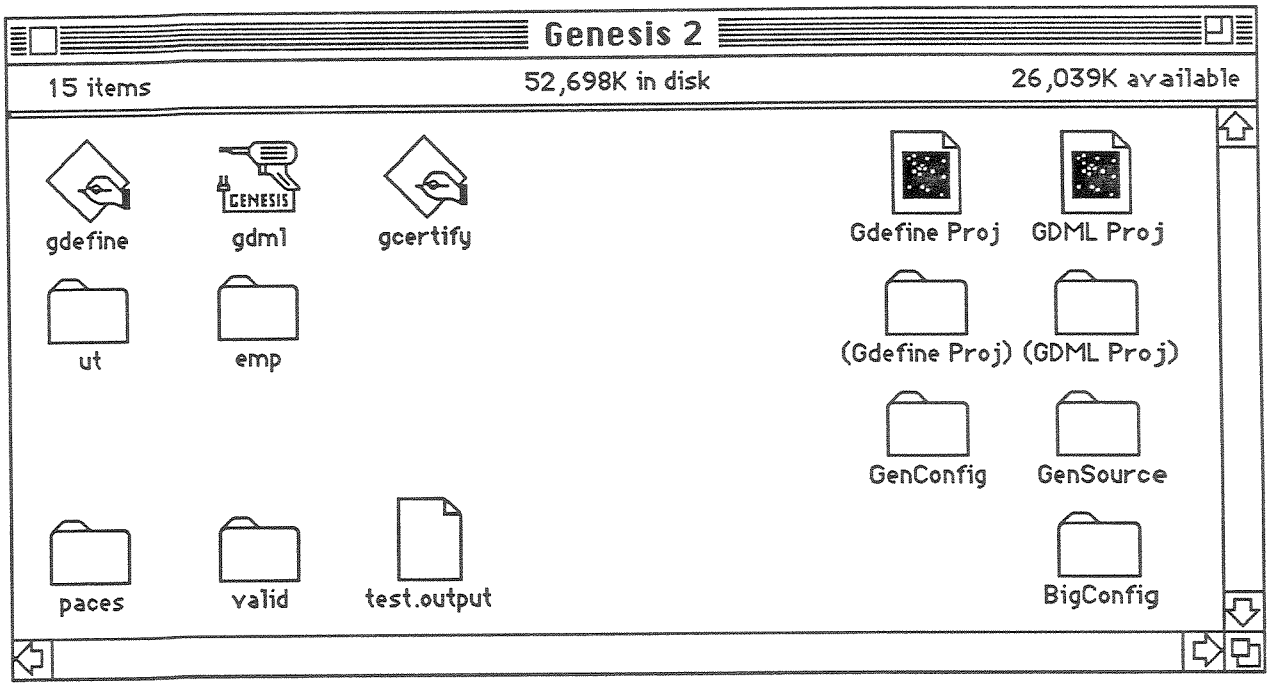
```
/*****************      RECOVERY  TYPES      ********************/

#define BFIM_RECOV                  1
#define NULL_RECOV                  0
#define SHADOW_RECOV                0
#define DBCACHE                     0
```

These commands will define Universals that rely on before-image logging. Altering the use (1) and nonuse (0) values, one can select different recovery implementations. Remember: only one entry should be set to 1; the rest must be zeros.

When you are compiling Universals, discard the genesis_config.h and jupiter_config.h files generated by DaTE and use those in BigConfig. Rename all other configuration files to "demo", as this is the name given in the BigConfig universals. (Actually, to change the name to something other than demo, open up genesis_config.h in BigConfig and edit the #define DB_NAME to whatever name you'd like). Other than this, building Universals is no different than building any other DBMS.

Phase 3 - Usage

The usage phase deals with the customization of DDL directives, creation of schemas, database loading, and database processing. To get started, recall the **Genesis 2** folder:



We are interested in the icons on the left-hand side. (With the exception of the **GenConfig** folder, the remaining files are no longer needed).

As a tour, we created the applications **Gdefine** and **Gdml** in Phase 2; they are the executables of our target DBMS. The application **Gcertify**, and the folders **paces** and **valid**, and the file **test.output** are used to validate **Gdefine** and **Gdml**. (More on this later). The **emp** and **ut** folders are two small databases provided with Genesis. The **emp** database consists of a single relation; the **ut** database has three highly interconnected relations.

Phase 3 begins by customizing DDL directives.

Customizing DDL Directives

The .OPT file generated by DaTE lists DBMS-specific directives on how to store different relations and links. DaTE assigns a name for each directive, but the name itself may not be syntactically correct or easily rememberable. Customizing DDL directives is, in effect, providing alternate names.

The names DaTE generates reference labels given to layer parameters or user-defined architectures. Any sequence of characters can serve as a name for DaTE. However, the current Genesis DDL only allows names to begin with a letter, followed by a sequence of letters, digits, or underscores (_). Illegal names must be repaired by editing.

As an example, open **Ingres.OPT**, which is in the **GenConfig** folder. You'll see the following text:

```
Row          Tag   Next   Type   Path   Fit   Name
#
  0           1     -1      1      0     -1    ingres.ss
  1           2      2      0     -1      1    hash_index
  2           4      3      0     -1      2    heap_index
  3           8     -1      0     -1      0    isam_index
  4          10      5      0     -1      0    isam_internal
  5          20      6      0     -1      1    hash_internal
  6          40     -1      0     -1      2    heap_internal
```

The only column of interest to us is `Name`. (Don't change any other entries!!). The label `ingres.ss` should be familiar; it is the name of the ingres storage system that we defined in Phase 1. We need to rename this label, to say '`ingres`', because it has an illegal character (i.e., the dot):

```
  0           1     -1      1      0     -1    ingres
```

All other names are acceptable as is.

Once the Name column contains legal names, further renaming can be motivated by examining the semantics of each label. Schema-defined relations that are to be stored via the ingres storage structure must be tagged with the option '`ingres`'. When there is only one storage system - as in our case - using explicit storage system tags is unnecessary. Tags are needed if relations could be mapped by several storage systems. (Recall that multiple storage systems are possible when a DBMS has supports multiple architectures). We'll illustrate tagging shortly.

The next three options - `hash_index`, `heap_index`, and `isam_index` - are directives to store index files in hash, heap, or isam structures. The last three options - `isam_internal`, `hash_internal`, and `heap_internal` - are directives to store data files in isam, hash, or heap structures. Again, we'll explain how they are used in schemas shortly.

We'll shorten the labels in rows 1-6 by dropping `_internal` suffixes and replacing `_index` suffixes with 'x', yielding:

```
Row          Tag   Next   Type   Path   Fit   Name
#
  0           1     -1      1      0     -1    ingres
  1           2      2      0     -1      1    hashx
  2           4      3      0     -1      2    heapx
  3           8     -1      0     -1      0    isamx
  4          10      5      0     -1      0    isam
  5          20      6      0     -1      1    hash
  6          40     -1      0     -1      2    heap
```

In general, any name for a label can be used as long as it is unique within the table and does not conflict with Genesis DDL reserved words. These words are listed below:

int	files	index
cstring	links	primary_key
vstring	database	ring_list
float	rpg	ptr_array
double	set	
byte	char	

Defining Schemas

Open the file **ut.schema** in the **ut** folder. It is shown below with labels in bold:

```
DATABASE ut {

FILES

     employees  {
        empno        INT                    primary_key;
        age          INT                   ;
        dept_name    CSTRING ( 20 )        ;
         name          CSTRING ( 22 )         primary_key indexed isamx;
      }  heap ingres;

     dept {
        deptno       INT                    primary_key;
        dept_name    CSTRING ( 20 )         indexed hashx;
        chairman     CSTRING ( 22 )        ;
      } isam;

     prof {
        profno      INT                     primary_key;
        prof_name   CSTRING ( 22 )         ;
        department  INT                    ;
      }  hash;


LINKS
     /* 1:n links */
     P_worksin    : dept.deptno = prof.department          /* ring_list */;
     E_worksin    : dept.dept_name = employees.dept_name   /* ring_list */;

     /* 1:1 or 1:0 link */
     P_empdata    : prof.prof_name = employees.name        /* ring_list */;
     P_chairdata  : prof.prof_name = dept.chairman         /* ring_list */;
     E_chairdata  : employees.name = dept.chairman         /* ring_list */;
}.
```

Three relations are defined plus five links. The syntax for relations is straightforward. Link syntax is <name> <colon> <join-predicate> <options>. Join predicates must be equality-based with no disjunctions. Links interrelate a pair of different relations. One is the **parent** and the other is the **child**. Parentage is conveyed by phrasing of the join predicate; the first relation referenced is the parent. Thus, in P_worksin, the dept relation is the parent and prof relation is the child.

DDL directives are options that can adorn relations, individual fields, and links. In addition to those in ingres.OPT, there are a few options that are reserved: **indexed**, **ring_list**, **primary_key**, and **ptr_array**. **indexed** is used to tag fields that are to indexed; **primary_key** tags fields that define the primary key of a relation; **ring_list** and **ptr_array** tag links to specify that their implementations are to be ring lists or pointer arrays. Note that only **primary_key** is provided to all Genesis-produced DBMSs. **index**, **ring_list**, and **ptr_array** are available only if their corresponding layers are present in the DBMS.

To see how these directives are used, look at the employees relation:

```
employees  {
    empno          INT                    primary_key;
    age            INT                   ;
    dept_name      CSTRING ( 20 )        ;
     name           CSTRING ( 22 )          primary_key indexed isamx;
 }  heap  ingres;
```

The field pair (empno, name) is declared to be the primary_key of employees. The name field is to be indexed, and its index file is to be stored in an isam structure. The employees relation is to be stored in a heap. The option 'ingres' adorns employees to tell **Gdefine** that employees is to be stored via the ingres storage system. (As mentioned earlier, the use of the 'ingres' tag in this example is not necessary). Normally, when one specifies a field to be indexed, one also must specify how the correponding index file is to be stored. If multiple options are available, but no option is given, **Gdefine** will make a choice and will report its choice during schema compilation.

Now look at the P_worksin link:

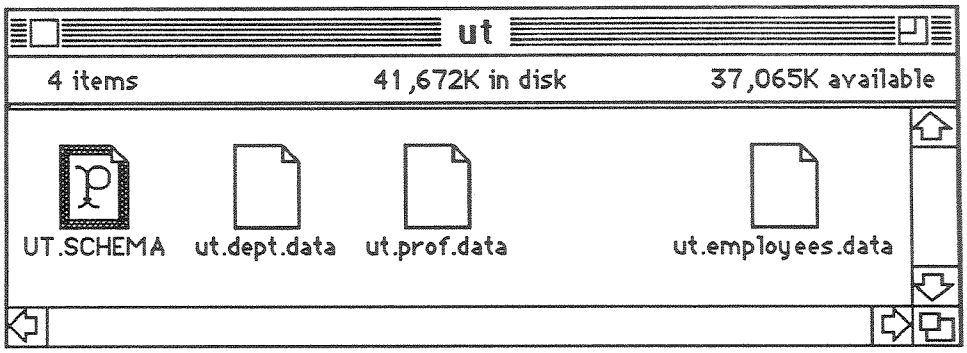
```
P_worksin    : dept.deptno = prof.department          /* ring_list */;
```

If the ingres DBMS supported ring lists, the **ring_list** tag would be placed as shown, but not within comment markers. The P_worksin declaration above is untagged. An untagged link tells **Gdefine** that the link must be implemented by a join algorithm.

> Note: If there are no link layers, the current version of **Gdefine** will still recognize links. It is only during Gdml execution that references to the link will be recognized as an error.

Defining Databases

All information about a database is stored in a folder labeled with its name. It must be in the same folder/directory as **Gdefine** and **Gdml**. As an example, open the ut folder:



The schema is present, along with three other files. These files, respectively, contain the raw data that is to be used to load the dept, prof, and employees relations. Their format is straightforward. Each line contains data for a record. Column values are separated by commas with no embedded blanks; strings are enclosed within quotes. The naming of raw data files is important. If s is the name of a schema and r is the name of a relation in a schema, then the name of the raw data file is 's.r.data'. Thus, ut.dept.data is the raw data file for the dept relation in the ut database.

When **Gdefine** is run, the following output is generated:

```
***** Creating Volumes *********

:ut:ut created

***** Creating Files ***********

employees created
$emp!name created
dept created
$dep!dept_name created
prof created
```

The names of each volume, conceptual file, and internal file that is created is listed. Internally generated files have names prefaced by $. Index file names, for example, take the first three characters of the conceptual file, followed by bang (!), followed by the name of the field. Thus $emp!name is the name given to the name index file of the employees relation.

Gdefine produces a large number of additional files that will appear in a database folder. These files are used by **Gdml** to know how to perform conceptual-to-internal mappings of relations and links in the database. When **Gdml** is run for the first time on a newly compiled schema, it will look for a raw data file for each relation, and will load that relation. Hell breaks loose if the file isn't found. Subsequent runs of **Gdml** will not invoke database loading.

Note: To dump a database, one simply has to execute a "select * from relation" and route the output to a text file. The format used in **Gdml** to output tuples is the same format it uses for raw data file loading.

Database Processing

A Genesis-produced DBMS can run subsets of SQL, QUEL, or both. The Gdml prompt 'SQL:' requests a SQL command; 'QUEL:' prompts for a QUEL command. Typing SQL will switch the DBMS to accept SQL commands; typing QUEL switches to QUEL commands. (Switching is possible only if a DBMS supports both interfaces).

A subset of SQL and QUEL is supported. This doesn't include, unfortunately, nested selects and aggregations. So too are join predicates within update and delete statements. (That is, update and deletion predicates are restricted to reference a single relation).

> Note: these restrictions were imposed simply to keep this version of Genesis tractable. There is no apriori reason that prevents full-blown SQL and QUEL to be supported.

Link names can appear in SQL predicates in place of their schema-defined join predicate. Thus, the following queries are identical:

```
select *                                    select *
from   dept, prof                           from   dept, prof
where  dept.deptno = prof.department        where  P_worksin
```

If aliases are used in an SQL select, link names cannot be used. (The reason is that link names are bound to schema-defined names of relations, not aliased names). This means that link names cannot be used in QUEL retrieves, because of the standard use of aliasing relations via RANGE-OF commands.

If a dml statement becomes too long to fit on a single line, lines can be continued by a backslash (\) carriage return. Although not shown, the first two lines of both of the above select statements were terminated by a \.

QUEL has been extended to support ORDER_BY clauses to order the output of retrieval statements. The syntax is the same as that for SQL. Another simple extension is the * feature for retrieval:

```
retrieve (E.*) where...
```

where 'E.*' is a shorthand for all fields of relation E.

Validation

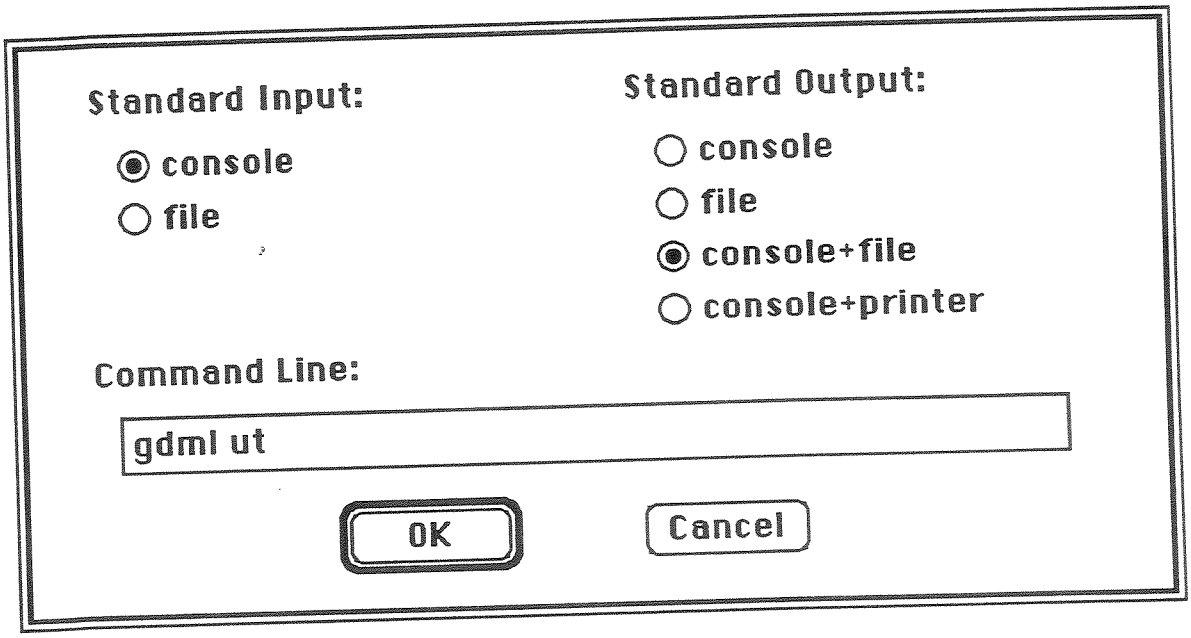
A set of SQL scripts has been developed to validate Genesis produced DBMSs against the emp and ut databases. For the emp database, the script is **file.paces**. For the ut database, three scripts are available: **link1.paces** tests multifile retrievals, **link2.paces** tests multifile updates, and **link3.paces** tests cyclic queries. These files are located in the **paces** folder in **Genesis 2.**

To run link2.paces, as an example, type in the statement:

SQL: source :paces:link2.paces

where SQL: is the **Gdml** prompt for an SQL command. Input will then be taken from link2.paces until the file has been exhausted. To validate the output of **Gdml**, it is necessary to route it both to the console and to a separate file. This can be done by selecting **console+file** as the standard output, and giving ut as the command-line input. (**test.output** is the file that we use).

Note: routing output just to a file, and not to the console, will generate slightly different output than routing to both a file and console. The difference is that routing to the console will introduce extra line feeds and carriage returns when a line becomes too long and wraps around.



The application **Gcertify** is a simple program that compares two files and determines if both are identical (OK) or not ('files are different'). The table below lists the paces files and their validated output files.

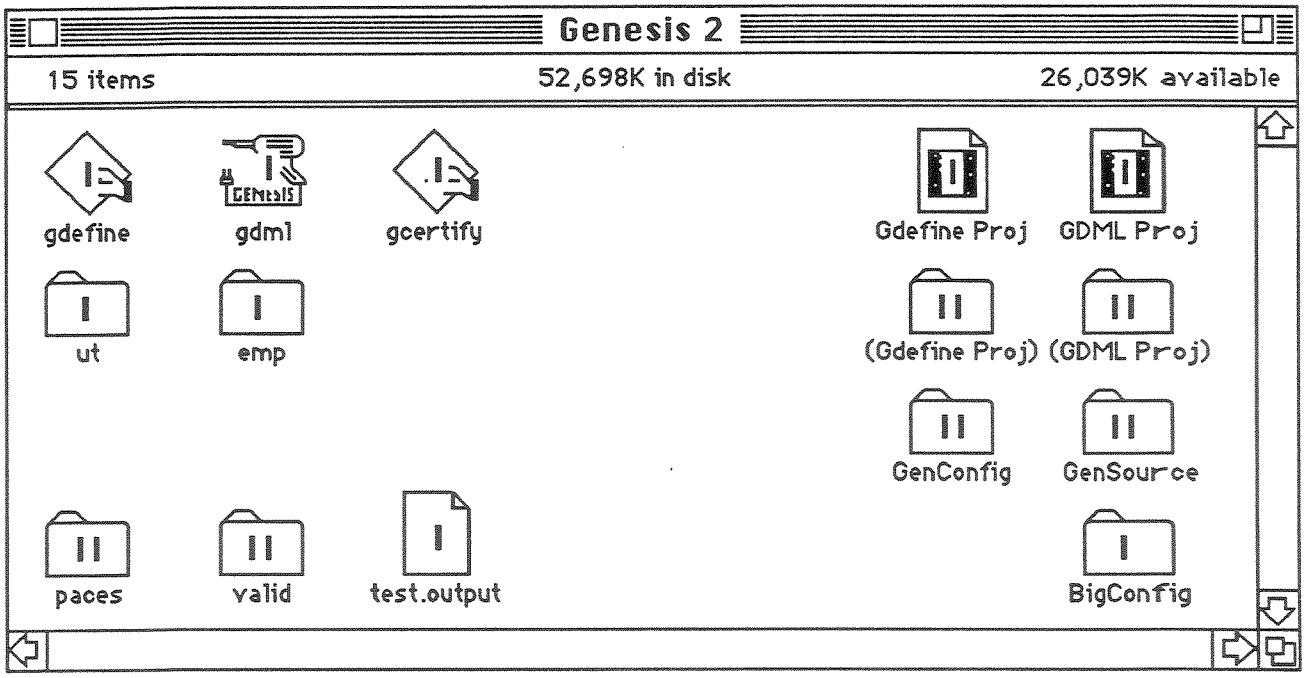
paces files	validation files
:paces:file.paces	:valid:file
:paces:link1.paces	:valid:link2
:paces:link2.paces	:valid:link2
:paces:link3.paces	:valid:link3

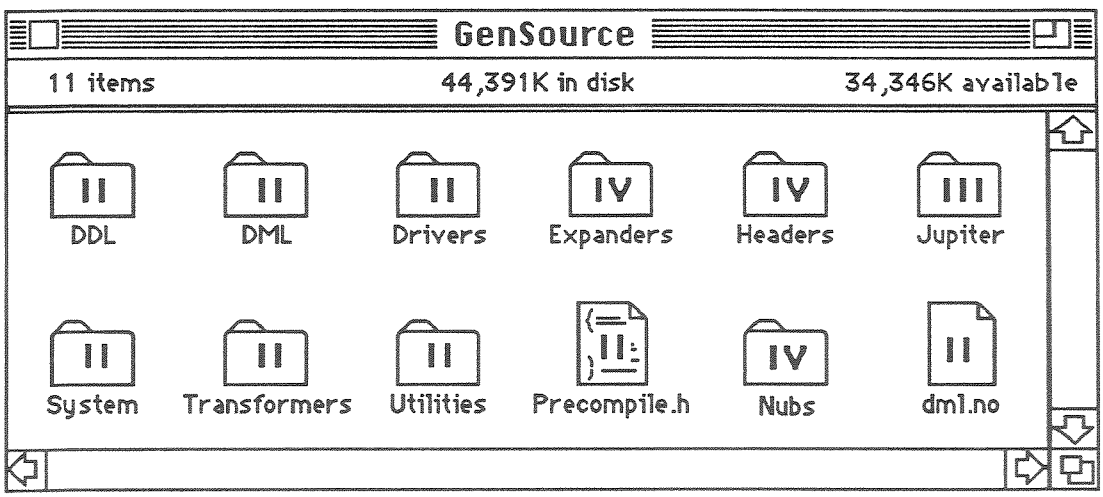
To validate the output of link2.paces, for example, run **Gcertify** with command line parameters test.output and :valid:link2.

Installing Genesis

Genesis runs on Mac IIs with 6-8M of memory. It's best with a 80M hard disk, as databases can eat up space very quickly. Genesis comes on four 1.4M disks. Mac IIc series can read these disks; original Mac IIs cannot. So if you are getting 'unreadable disk' errors, then most likely your drive can't read 1.4M disks.

The disks are numbered, and should be loaded in ascending order. Disk #I contains the DaTE folder and the Genesis 2 folder. The Genesis 2 folder is incomplete; its contents are (partially) completed with the addition of the files in Disk #II. Disk#3 and Disk#4 contain the remaining files/folders that are to appear in the GenSource folder of the Genesis 2 folder. The figures below show the individual assignment of each folder and the disk number (I, II, III, IV) on which the folder is stored. Try to keep the arrangement of folders, as shown below:





Some final comments. In case you lose dml.no, you can regenerate it by running the Nubs compiler (application nc in the Nubs folder of GenSource) with the command-line arguments "dml.nu dml.no". This causes nc to translate the file dml.nu (which you'll find in the dml folder) into the file dml.no.

GenConfig contains Universals and the configuration files for Ingres. So if you proceed to compile Gdefine and Gdml out of the box, you'll have an imitation of University Ingres. Keep in mind that when you run Gdml, it will present you with an SQL prompt. That's because you're running a Universal which presents both SQL and QUEL.

The Genesis Disks contain no Gdefine or Gdml executables. You'll have to produce these yourself. (In their place on Disk #I are some empty text files. They're present only to show you where the executables will be placed once they are created).