Persistence Services

of MC-3020

Design and Architecture Notes

Preliminary Draft

considerations for non-volatile storage support in the MC-3020 model compiler

ROX Software, Inc.

Persistence Services: of MC-3020; Design and Architecture Notes; Preliminary Draft; considerations for non-volatile storage support in the MC-3020 model compiler

Revision History

Revision 0.1 19 Feb 2004 Revised by: cds
first draft
by ROX Software, Inc.

Published February 2004

This document explains persistence.

Table of Contents

List of Figures

List of Examples

Chapter 1. Persistence

This chapter describes persistence services for MC-3020.

1.1. Introduction

Persistence capability in a model compiler refers to the ability of the model compiler to allow dynamic data to persist across the boundary of a power cycle. The value of a persistent attribute written before power disconnect or power loss will be restored when the power returns.

MC-3020 does not support persistence of dynamic data in versions 3.3 and earlier. This section outlines requirements and operation of persistence support.

1.2. High-Level Requirements

1.2.1. Non-volatile Storage Technology

There are a great number of options for non-volatile storage (NVS) technologies in the embedded control world. Applications exist which use more than one non-volatile storage technology within the same system. It is required that modularity exist at the interface between general persistence services and the driver level code storing and retrieving data from non-volatile storage. A bridge is defined to allow MC-3020 users to build or replace this driver layer.

1.2.2. Granularity

Different applications need to persist different amounts of data. The ability to persist at the class and domain levels allows for both large and small amounts of persistent data. MC-3020 supports persistence at both class and domain level.

1.2.3. Balance

Some approaches to persistence can add tremendous complexity to the model compiler. Overly simplistic approaches to persistence support can pollute the application analysis. MC-3020 strikes a balance that makes sense for the embedded applications to which MC-3020 is best suited.

1.3. Operational Overview

1.3.1. General Scenario

Persistence services for MC-3020 are light weight and flexible in terms of non-volatile storage technology. The persistence services are broken between two domains, persist and NVS.

The persist domain performs the commit and restore operations which commit instances to non-volatile storage and restore instances from non-volatile storage respectively. The persist domain keeps track of which instances have been persisted, which links have been persisted and manages keeping the links synchronized with the instances during the power-up system initialization.

The NVS domain supplies a rudimentary but functionally complete persistent data storage and retrieval interface. The interface has characteristics of both a database and of a file system. The interface is simple enough to provide flexibility in the application of different non-volatile storage technologies.

MC-3020 persistence does not specify or depend upon a specific persistent storage technology. Therefore, only a standard interface (bridge) is defined. The user may deploy whatever available technology desired behind the bridge to the NVS domain. (A sample implementation is supplied for use as the architect desires.)

The domain chart below shows the bridge operations made visible to the application. `commit` is the primary function used by the application.

The architecture domain (MC-3020) automatically performs the `inserts`, `updates` and `deletes` to shuffle instance data between the RAM based collections of the application and the non-volatile persistent storage. Additionally, MC-3020 performs the `restore` operation at power-up time.

Other domain interfaces are exposed and may be used at the discretion of the analyst.

1.3.2. Manual and Automatic Commit

A *manual commit* occurs when the user forces a commit of instance and link data to non-volatile store by synchronously invoking a persist domain function (e.g. `persist::commit()`). An *automatic commit* occurs when a commit is initiated by the software architecture "behind the scenes" based on pre-defined policy.

MC-3020 supports manual commit operations and does not support automatic commit.

1.4. Assumptions

The implementation and deployment of the Non-Volatile Store (NVS) domain is the responsibility of the user. MC-3020 supplies one or more sample implementations of this domain to serve as design examples and source code "head starts". ROX Software, Inc. and Project Technology, Inc. will develop and collect additional samples over time and make these available on the internet. However, with the variety of non-volatile storage technologies available and broadly differing platform requirements this deployment is left to the user for the purposes of flexibility.

Class instances and relationship instances (links) are stored in non-volatile storage. Their types are kept distinct from the rest of their instance data.

1.5. Limitations

Only class instances and links between persistent classes are stored/retrieved from NVS. Events and timers are not persisted.

1.6. Coloring

1.6.1. Coloring Persistence

Persistent classes retain the values of their attributes across power cycles. This includes the current state attribute. Newly created and updated classes are "backed up" to non-volatile storage. At system start-up time, any classes stored in non-volatile storage are restored before other application initialization occurs. Individual classes can be colored to be persistent.

Use this color to mark a class as persistent.

```
TagPersistentClass(string ss_name, string class_key_letters);
```

```
TagNonPersistentClass(string ss_name, string class_key_letters);
```

Where the input parameters are:

ss_name

name of subsystem

class_key_letters

keyletters of the persistent class

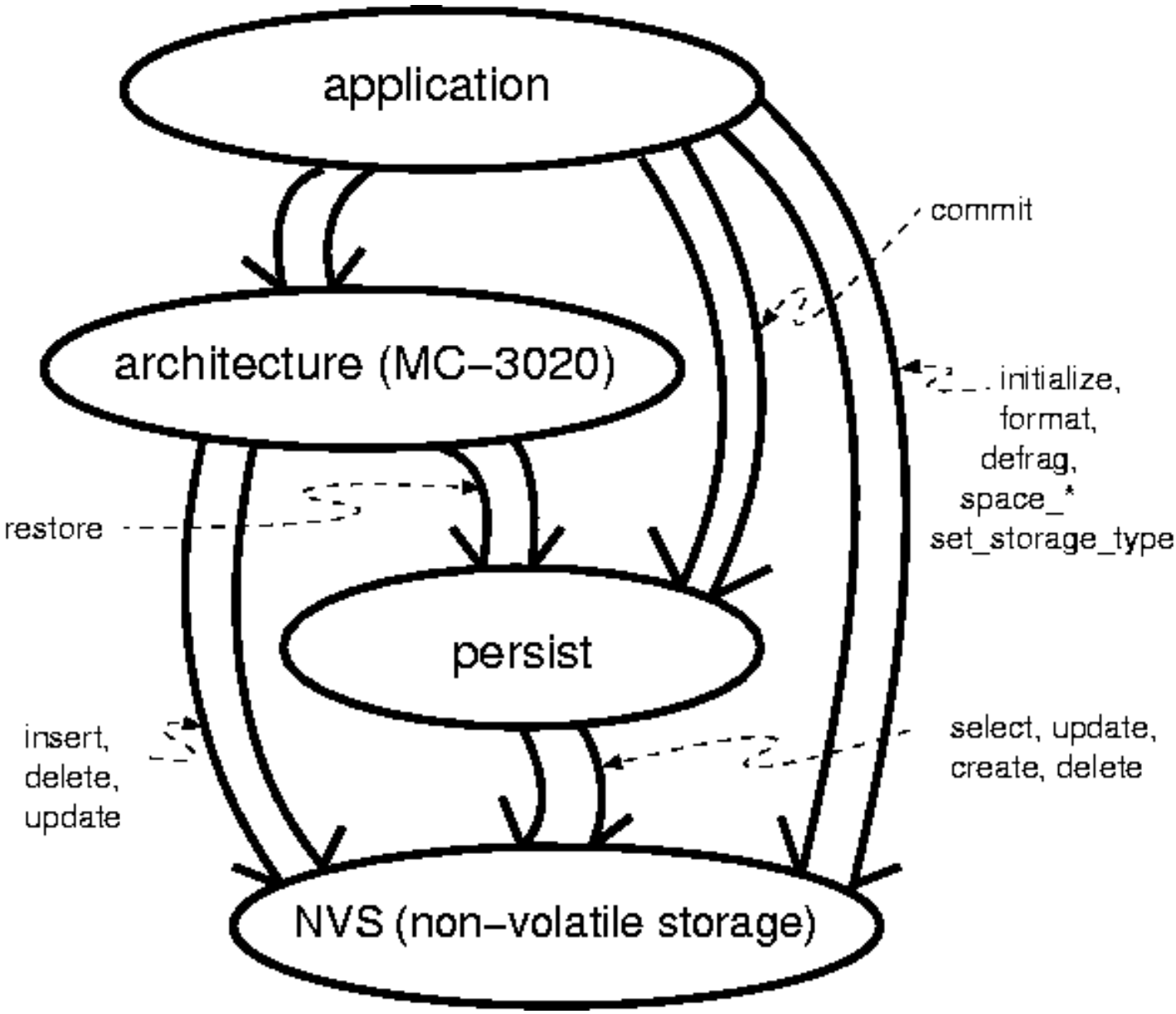
Example 1-1. Marking (Non-) Persistent Classes

```
.// To mark as persistent a specific class, use "" for "ss_name"
.// and provide the class key letters in "class_key_letters".
.invoke TagPersistentClass( "", "MP" )

.// To mark all classes in the subsystem as persistent, provide
.// the subsystem name for "ss_name" and "*" for
.// the "class_key_letters".
.invoke TagPersistentClass( "TRACKING", "*" )

.// To mark all classes in the domain as persistent, use "*"
.// for "ss_name" and "class_key_letters".
.invoke TagPersistentClass( "*", "*" )

.// To mark as non-persistent a specific class that had previously
./  been marked as persistent, use "" for "ss_name" and
```

```
.// provide the class key letters in "class_key_letters".
.invoke TagNonPersistentClass( "", "ASN" )
```

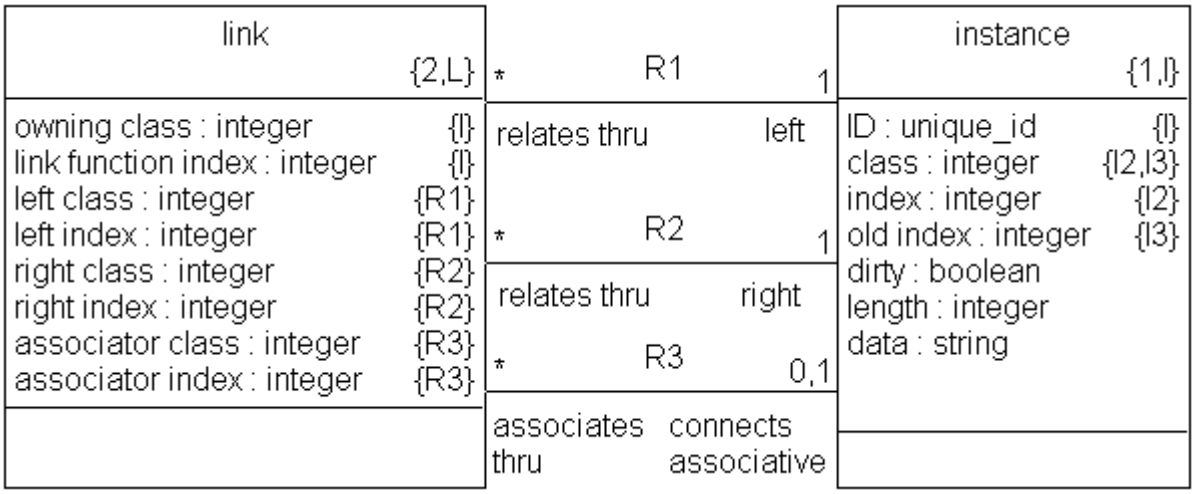
1.7. Analysis

Figure 1-1. Persistence Analysis Domain Chart



1.7.1. Persist Domain Data Analysis

Figure 1-2. Persistence Class Diagram



1.7.2. Persist Domain Functions

The commit function is called to indicate to the domain that the application wants to commit instances of classes and associations to be committed to non-volatile storage.

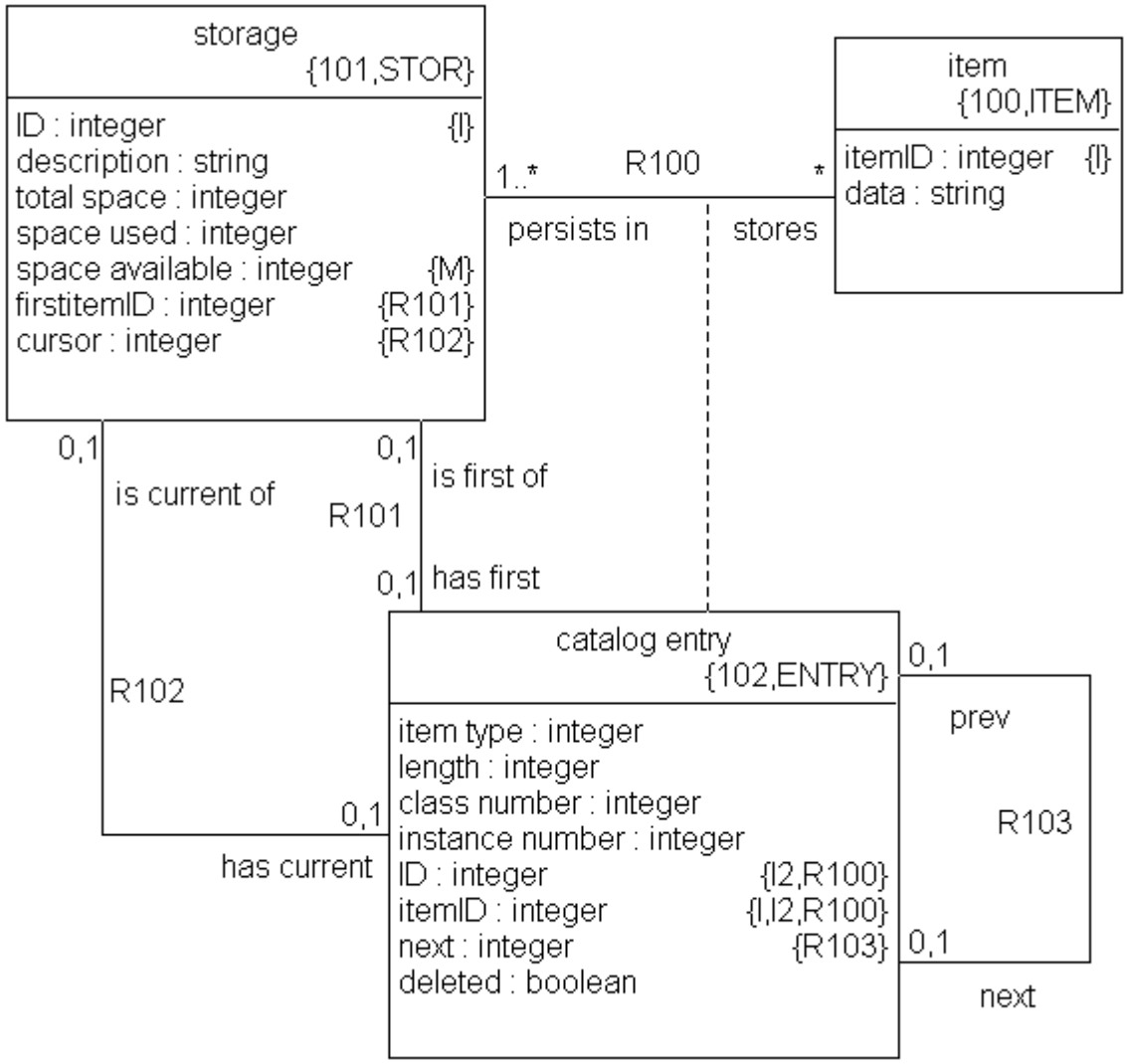
```
integer commit(void);
```

The restore function is called by the architecture during bring-up after a power cycle. The restore function causes the classes contained in non-volatile storage to be read from store and written to the instance collection list.

```
integer restore(integer class);
```

1.7.3. Non-volatile Storage Domain Data Analysis

Figure 1-3. Non-Volatile Storage Class Diagram



1.7.4. Non-volatile Storage Domain Functions

This function adds items to the store. There are four Input arguments. The first the key to the item. This key is a unique identifier for the item being inserted (added to the store). The length is an integer representation of the length of the data string found pointed to by pointer. The type is the class type in integer form. A return code provides status on the insert.

```
integer insert(integer key, integer length, string pointer, integer type);
```

The update function searches for a record in the store the same way that select does. When (and if) a record is found, the new data of length length pointed to by pointer is written into the store over the existing item.

```
integer update(integer key, integer length, string pointer, integer type);
```

The select function searches the store for a specific item with a key and type matching those given as input arguments. If a record is found that matches, it is copied into the data buffer pointed to by pointer. The length argument provides the amount of buffer space available on the calling side. The actual length of the returned record (if one is returned) is provided in the return value.

```
integer select(integer key, integer length, string * pointer, integer type);
```

Delete searches the store for a item matching the key and type given as arguments. When found, the item is marked as deleted and will not be readable from the store.

```
integer delete(integer key, integer type);
```

The space_available function returns an integer representing the number of bytes not being used in the store. This number of bytes will actually hold fewer bytes due to the overhead of item meta-data (key, type, etc).

```
integer space_available(void);
```

The space_used function returns the number of bytes currently stored in the store.

```
integer space_used(void);
```

The space_total function returns the overall size in bytes of the non-volatile store.

```
integer space_total(void);
```

The initialize function resets the internal counters of the non-volatile store. No data is written or changed in the store. This function is to be called at power-up to prepare the store for access.

```
integer initialize(void);
```

The format function erases the non-volatile store (NVS). It is only to be called when it is desired that a new NVS be cleared and prepared for writing for the first time.

```
integer format(void);
```

The defrag function coalesces deleted records together and written records together. This allows for small fragments of free storage to be collected into a single contiguous free piece of storage. This function will typically take significant time to run. This time is a function of non-volatile storage technology.

```
integer defrag(void);
```

To use more than one non-volatile store, the set_storage_type function can be called to switch the current store. The desired store is passed as the only argument.

```
integer set_storage_type(integer ID);
```

Next provides a way to cycle through reading each item from the store one and at a time. The next function returns the item currently being pointed to by the cursor (maintained inside the store). The buffer space available on the calling side is passed in as the length argument. If sufficient space is available in the buffer, the next item data will be copied into the given buffer pointed to by pointer. The length of the written data is returned as the return value. Also returned are the values of the key and type. The initialize function resets the internal cursor to the first item in the store.

```
integer next(integer * key, integer length, integer * pointer, integer * type);
```

1.8. Cost Modeling

Persistence services have cost in terms of execution speed, instruction store space, data memory and non-volatile storage space. This section enumerates those costs in as detailed a manner as practical.

1.8.1. Data Memory Cost

To manage the bookkeeping of persistent instances, additional data is maintained. The following additional storage is used for each instance of each class.

1. The *old index* is kept for each instance. This value represents the index into the array of instances. The value is the index of the class instance when it was stored into NVS. This is required during the restore operation only but takes up space none-the-less.
2. The *dirty bit* is required to keep track of which instances have been changed since they were last flushed out to NVS.
3. list of pointers to link functions
4. list of class numbers

1.8.2. Instruction Store (Code Space) Cost

TBD

1.8.3. Cost in Speed

TBD

1.8.4. Non-volatile Memory Cost

TBD