Software Versioning

Procedure

CONFIDENTIAL AND PROPRIETARY

Hart InterCivic, Inc.	
Part Number: 1001070	REV: C.00
Part Name: Software Versioning Procedures	
File Name: Software Versioning Procedure - 1001070 C00.docx	Page 1 of 14

Change History

Version	Date	Author	Description
0.1	7/11/06	[redacted]	Initial Creation
0.2	7/21/06	[redacted]	After initial comments from review members
0.3	7/28/06	[redacted]	Updates after comments from Marco Tabone
0.4	12/08/06	[redacted]	Updates after comments from Bill Jordan
A	2/7/07	[redacted]	Final edits
B	3/15/07	[redacted]	Updated the “Proprietary and Confidential” statement
C.00	08/17/2012	[redacted]	Update to current processes

Table of Contents

SECTION 1. INTRODUCTION

OVERVIEW

Software versioning procedures fall into the category of a business process required to be clearly defined within an organization. Without a well-defined process it becomes increasingly difficult for an organization to support, maintain, trace and account for the applications that are released to its customers and to identify customers that need to be notified when updates or upgrades are available for them, specifically updates that improve system behaviors.

The focus of this document is to define consistent versioning throughout different parts of the software lifecycle and to minimize the possibility of incompatibility between separate groups within an organization. Since the nature of this subject has so many relationships to other parts of the software lifecycle, this document eludes to them without a fully defining each topic. It is assumed that the reader has a reasonable understanding of the following concepts, the software lifecycle, software lifecycle applications, version numbers, build numbers, automated software builds, production builds, version control, version control practices, software compatibility, archiving software builds and several Software Quality Assurance processes.

DEFINITIONS

Software - means machine-readable instructions and data (and copies thereof) including middleware and firmware.

Release - means any software, related updates or upgrades, licensed materials, user documentation, user manuals, and operating procedures that have been or will be made generally available to entities outside of Hart.

Version - is a state of an object or concept that varies from its previous state or condition. The term "**version**" is usually used in the context of computer software, in which the version of the software product changes with each modification in the software, the “version” is a specific “revision” of the software.

Build label – An entity that exists within a reversion control system that describes the state of a software project at some point in time and provides access to a manifest of files and directories such that the state of the project can be recreated from the label.

SECTION 2. VERSION & RELEASE

OVERVIEW

Versions and Releases are denoted using a quadruplet of integers to indicate Major, Minor, and Revision (Rev); Build numbers are for internal tracking and verification of the software build process and will not be visible to customers are part of the software version number. The rationale and rules to be used in assigning a version number are addressed later in this document.

<MAJOR>. <MINOR>. <REVISION>.<*BUILD*>

The above standard notation will be used to express the version. Each integer section is called a segment. Each segment is divided from other segments by a decimal point and the segments are described from left to right. For instance, we would use segment 1 to describe the position of the <MAJOR> value in the quadruplet or version. Each section of the quadruplet is referred to as a level or number (e.g. Major release level 2 or Build number 599).

Example:
"2.4.7.*1333*"

The version above is referenced as version 2.4.7

Where: Major release is 2, Minor release is 4, Rev release is 7 and Build is not used except for internal tracking/deployments, e.g. SQA validation.

Any child version levels will be reset when their parent level is incremented or reset. Child levels will always be reset to 0. In cases where a version level is incremented the value will be considered to be an integer and incremented by 1.

Example:

Reported version	Original version	CASE 1: Next major version	CASE 2: Next minor version	CASE 3: Next Rev version	CASE 4: Next Build
Internal	2.4.7.3214	3.0.0.5	2.5.0.111	2.4.8.1234	2.4.7.4321
External	2.4.7	3.0.0	2.5.0	2.4.8	2.4.7

Assuming that the current version is 2.4.7, the table above describes the outcome of creating the next major, minor, Rev, and Build versions.

Explanation of the preceding example:

Case 1 - Describes the creation of a new Major version where the Major number is incremented by 1, from major level 2 to a value of 3. The effect of this change is that the

Minor level is reset to 0, the Rev level is reset to 0, and the Build is reset to 0, because they are all children of the major level.

Case 2 - Describes the creation of a new minor version where the Minor version number is incremented from 4 to 5. Both the Rev and Build numbers are reset because the Minor level was incremented, the Rev level is reset to 0, and the Build is reset to 0.

Case 3 - Describes a new Revision that may potentially be released to manufacturing. When the Rev number is incremented from 7 to 8, the Build number is reset to 0, because the Rev level was incremented

Case 4 - Describes the next Build. Here no levels are reset because the Build level has no child levels.

The following example describes how to reference versions based on the values contained in the integer quadruplet. If the patch level is 0 and the minor level also has the value of 0 then it is not required that the minor and patch levels be spoken in casual speech. On the other hand when this information is included within an application, such as an about screen, all levels including the hot fix number must be present to properly identify the software release. This is important when both reporting and replicating defects for resolution.

Example:

- 3.0.0.5 – Would be reported as “version 3.0.0” by the software
- 3.0.1.8 - Would be reported as “version 3.0.1” by the software
- 2.5.0.6 - Would be reported as “version 2.5.0” by the software
- 2.4.8.1 - Would be reported as “version 2.4.8” by the software

MAJOR RELEASE

A **Major Release** is a full product release of the software. It generally contains new customer-facing functionality and represents a significant change to the code base comprising the software product or family of products, or is used to represent a significant marketing change or direction in the product. Major version numbers are generally incremented by the product management team, and generally are accompanied with a new marketing push, or to communicate a significant improvement to the product. The major version number is identified by the digit or set of digits to the left of the decimal point (e.g., x.0, segment 1 with a placeholder value of x).

General:

- The major version number will be unique for system feature sets; a major release may contain minor, release candidate, and/or build modifications of the major release.
- **May** be accompanied by a branch of the code base within version control for further minor and/or release candidate development.

Scope:

- Any change to the code base that prevents backwards compatibility (e.g. Addition or Removal of features, changes in the DB schema or API commands).
- Any new functionality that is customer facing.
- Any large marketing push that accompanies the product and redirects the product.

Example:

- 2.x, 3.x, etc…

Frequency:

- Market driven

Audience:

- New customers.
- Existing customers with qualifying contracts
- Existing customers desiring to upgrade to the new feature set

MINOR RELEASE

A **Minor Release** of the software may be comprised a rollup of several branched releases, enhancements/extensions to existing features or interfaces driven by internal or external requirements, external requirements could be driven by enhancements to meet new sales area (State specific features or rules), internal requirements could be enhancements aligned to a new marketing push. The minor version number is identified by the digit or set of digits to the right of the decimal point separating it from the Major Release number (e.g., 2.x, x indicates the Minor Release placeholder).

General:

- The minor version number should always be unique, unless the release is a Revision or a Build Release.
- **Must** be accompanied by a label in version control.
- **May** be branched in version control for further patch development.
- Any minor releases of a particular major release are considered to be a part of the latest major release (e.g. 2.4, 2.5 and 2.6 are all part of major version 2.0).

Scope:

- Minor enhancements and features that do not affect compatibility with its associated major or current minor releases.
- New features or functionality that does minimally effect the interfaces
- Addition of new features or functions to meet a new sales area that doesn't affect existing customers of the release.
- To accompany or communicate a new marketing initiative.
- Error corrections and maintenance.
- A rollup of branched releases.

Example:

- 2.4, 2.5, etc…

Frequency:

- Enhancement and Market driven

Audience:

- New customers.
- Existing customers with qualifying contracts.
- Existing customers desiring to upgrade to the new feature set

REVISION RELEASE

A **Revision (Rev)** is a build of all or part of the software that is initially distributed to an internal audience, specifically software quality assurance, for software validation. If the Rev is successfully validated and accepted, this version is "released" to manufacturing. If defects are found that prevent the Rev from successfully being validated and prevent the release to manufacturing, the Rev value will be incremented prior to the next validation cycle.

General:

- The Rev version number should always be unique, unless the build is a SVN build of the original Rev release.
- Any Rev build of a particular major or minor release are considered to be a part of the latest minor release (e.g. 2.4.5, 2.4.6 and 2.4.7 are all part of minor version 2.4 and major version 2.0).
- **May** be branched in version control if SVN builds are needed.

Scope:

- Releasing build to SQA for validation

Example:

- 2.4.1, 2.4.2, etc…

Frequency:

- As necessary, but typically every 1 to 3 months.

Audience:

- Existing customers with qualifying contracts.

BUILD RELEASE

A **Build Release** is a build of all or part of the software distributed to an internal audience, these releases should have targeted feature enhancements and issue resolution documented to allow testing in the targeted/specific areas where the changes were implemented. The Build version number is denoted by a fourth digit or set of digits (e.g. 2.4.7.x, segment 4 with a placeholder value of x), corresponding to an internal build number.

General:

- The Build version number **must always be unique**.
- Any Build of a particular major, minor or patch release are considered to be a part of the latest Rev release (e.g. 2.4.5.1 is part of patch version 2.4.5, minor version 2.4 and major version 2.0).

Scope:

- Resolves a particular defect, typically of a critical Severity level with no viable workaround.

Example:

- 2.4.7.1, 2.4.7.2, etc…

Frequency:

- Extensively used during internal developement.

Audience:

- Internal developers
- Internal verification

SYSTEM RELEASES CONTAINING MULTIPLE PRODUCTS OR COMPONENTS

Quite often several products or components are bundled into a System Release to provide an expanded set of tools or functionality. There are multiple ways that the parent child relationship can be created.

The preferred methodology for new systems is for each child product to have a supplementary version field that identifies it as part of the system release version. For instance a dll or exe file might contain multiple version fields in the binary header data, one identifying the parent system version and one identifying the dll version.

A second method is to manage the individual child products versions separately allowing for unrelated versions that follow the guidelines set forth in this document. In such cases, there is no need to make or modify the individual component versions such that they all reflect one unique release value. Instead, there will be a parent system version or release that acts as a wrapper and contains all of the individual child product. To accomplish this, the parent version must follow the same guidelines described herein. By binding the parent to these guidelines we can manage them in the same general way as their children with respect to the software lifecycle.

Example: “Verity 1.0 = 1.0.23”

Child Product	Child Version
Verity Build	1.0.4
Verity Count	1.0.13
Verity Central	1.0.7
Verity Relay	1.0.4
Verity Scan	1.0.1
Verity Controller	1.0.3
Verity Touch	1.0.3
Verity Touch Writer	1.0.3

Any sub-projects of a child project should also reference only the top-level parent or system version to avoid a cascade of version fields. For example, using the examples from the diagram above, if Verity Touch consisted of 2 sub projects, hardware and software, then they would only reference Verity 1.0 not Verity Touch 1.0.3.

Another way to track to the child project versions in a system would be to add a licensing and/or provisioning application that monitors the status and configuration of the system including the version of any child applications. This would require a network service or daemon where each child application would then need to register itself before it could be used within the system.

COMPATIBILITY

We define "**source compatible**" to mean that an application will continue to operate without error, and that the semantics will remain unchanged with respect to a previous version.

Applications that operate against a particular version or release should remain source-compatible against later minor versions. However, for example if an application that is part of a parent release contains significant changes or enhancements it may no longer build or operate against a previous major or minor version of the parent release.

We define "**binary compatible**" to mean that a compiled application can be linked (possibly dynamically) against an existing library and continue to function properly.

Similar to source compatibility, an application that has been compiled against a particular version should continue to be linkable against later versions, although it is possible that an application will not be able to successfully link against a previous minor version.

Original Version	New Version	Compatibility
2.4.5	3.0.0	No, compatibility with different Major versions is not guaranteed.
2.4.5	2.5.1	Yes, compatibility with previous Minor versions is guaranteed.

2.4.5	2.4.7	Yes, compatibility between Rev versions is guaranteed.
2.4.5.1	2.4.5.2	No, compatibility between Builds is not guaranteed.

SECTION 3. VERSION CONTROL

The ability to deliver major, minor, release candidate, and build releases necessitates a version control system that provides access to the source code and state of a software project at any point in time throughout the software lifecycle. It is very probable that, somewhere within the lifetime of any software product, a release candidate will be needed to address issues with a previous release. Thus, it is imperative that all software projects maintain a procedure that facilitates the ability to recreate and rebuild historical releases. Since most currently available Version Control and Lifecycle Management systems support branching and labeling, as well as branching from a label, this section assumes that an appropriate version control system is available.

There are two common philosophies when it comes to branching, “Branch early and often” or “Branch only when necessary”. Using the first methodology, “Branch early and often”, typically means that there will be at least some parallel development in both of the branches. To make this methodology successful, a strict synchronization and merge process is required. The responsibility then falls squarely on development to maintain the delicate balance between the two branches (often called dual maintenance). Best intentions aside this methodology introduces a great deal of error as well as a lot of overhead to maintain the integrity of the individual branches. Instead, we will follow the “Branch only when necessary” model whenever possible in order to minimize that scenario. This philosophy relies on the earlier assumption that the available version control system supports labeling and also branching from a previously defined label, but it also helps to avoid several pitfalls related to the other methodology.

Guidelines for Labeling:

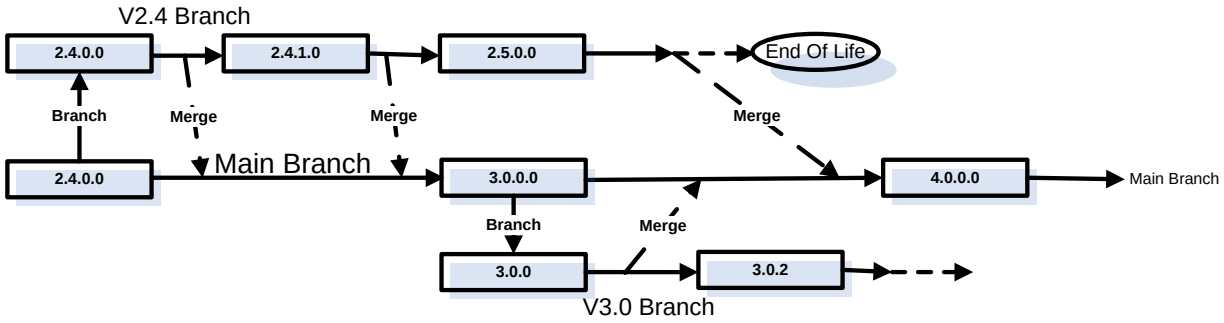
- Generate a label for every software build that will allow the software to be accurately rebuilt at any point in the future.
- Use a label name that appropriately represents your product and environment, see Section 5: Build Labels, for more details regarding label naming conventions.

Guidelines for Branching:

- Only branch when necessary.
- When creating new branches use a name that describes the product and the major and minor versions (e.g. “Verity 2.3”).

The following diagram is an example of how to use branching when branching is required.

Note: Solid lines indicate that a branch exists or has been created. The dashed lines indicate potential branches that could be created at any point if the need arose.



The example above starts with the initial branch of the 3.0.0 source code. The next branch is a Revision level branch of the 3.0.1 source code, which is later merged back into the 3.0.0 branch. Revision 3.0.2 is also branched and then merged back into the 3.0.0 branch. The 3.0.0 branch is then merged into the Main branch. Please note that the merge encompasses all the Revision branches created off of the 3.0.0 branch such that everything is merged back into the Main branch before the next minor branch is created. There may be times when a fix must be added to a previous branch after a merge has already been completed. If, for example, we needed to create 3.0.3 Revision release then the changes would need to be merged into the 3.1.0 branch and the Main branch.

SECTION 4. BUILD ARCHIVING

Build archiving is used to describe several levels of accessibility and is separate from the concepts described in the Version Control section. Builds are typically separated by promotion state based on their status as defined by Software Quality Assurance (SQA) group.

The promotion states include:

- Development State – Development builds (a.k.a. Nightly or Daily builds)
- Quality Assurance State - QA builds (a.k.a. Stable builds)
- Production State - Production builds (a.k.a. General Availability builds)

Software builds are typically done on a nightly or daily basis and they are called Development builds, but this is not a hard rule. Builds in the Development state have not been tested beyond a basic sanity test or automated regression test and are most often used by development for testing defect resolution and feature enhancements. Any builds

that are tested by the SQA group and pass through standard testing procedures will then be promoted to either the Quality Assurance state or the Production State as a Revision. Only builds that are slated to be released for General Availability should exist in the Production State.

To maintain traceability and provide accessibility network folders should be created to store the builds for each state listed above.

Any builds that have been determined by SQA to be valid for release to customers (i.e. Production builds) should also be archived as hard copies. This should typically include 3 CD's or DVD's, 1 for the customer, 1 for Hart escrow and 1 for SQA.

SECTION 5. BUILD LABELS

A build label is a label assigned to every build of the application within a branch of the code and it identifies the source code contained in each build. Build labels are a common source control mechanism to trace and/or recreate the release of an application. There are at least two schools of thought regarding the naming conventions, format and layout of build labels.

The first is Branch Naming and typically appends a sequentially generated build number to the branch name. The branching diagram above illustrates three maintained branches of the code: Main line, V2.4 Branch, and V3.0 branch. Given the mainline branch of the product line a project may develop internal code names to represent major branches of the application. Since a major and minor release name is impacted by corporate, contractual, and marketing demands you may find it better and less confusing to developers and test to associate a major release with a development code name. (i.e.: Ferrari was the development code name for Windows XP) With this approach it allows the company to restructure a release or rename it without impacting our build labeling process. Also, if product management or marketing decides restructure a release into a couple of minor releases instead of one major release; again it doesn't affect the build labeling model. So given these conditions, the following build labels structure is recommended.

The second naming convention is called Release Naming which parallels the target release version. This method also appends a sequentially generated build number to the label. The benefit of this convention is that there is always a direct and easily identifiable relationship between the build label and the target release version.

Branch	Internal Name Based on Product	Example Build Labels
2.4	Verity 2.4.7	2.4.7.111
3.0	eRegistryV3.0.0	3.0.0.121
4.0	ePollbook 4.0.0	4.0.0.333

The release name can be used throughout the software release process to maintain a high level of traceability between separate elements in the software release process. For example, you can easily relate a build label, network build path and a final CD Volume label using this naming convention.

Build Label:	2.4.7.378
Network Dir:	\\<some network location>\staging\ 2.4.7.378
Escrow CD Volume Label:	2.4.7

While this naming convention may appear to be less flexible in the face of changing market requirements, a configurable build label format within your automated build scripts can easily rectify this situation.

TIP: Avoid using spaces within names of files, folders, labels or anything else that may be accessed by an automated build script or process.

Please choose the most appropriate solution for your environment and then apply it throughout your software development process.