GNU Enterprise Forms

Developer's Guide

Copyright

Authors

Jason Cater

James Thompson

Additional Contributors

Reinhard Müller

Feedback

Comments and suggestions on ways to improve this document can be directed to: gnue@gnu.org

Acknowledgments

This book was created using OpenOffice.org.

Revision

Published July 2007. Covering GNU Enterprise Forms Release 0.6

Contents

Introduction
- Overview
  - What is GNUe Forms?....6
  - What GNUe forms is not....6
  - Features of GNUe Forms....6
- Key Concepts
- Anatomy of a GNUe Form
  - Options....7
  - Parameters....7
  - Datasources....7
  - Logic....8
  - Blocks....8
  - Fields....8
  - Layout....8
  - Pages....8
  - Labels, Entries, Buttons....8
  - Triggers....8
- A Sample GNUe Form Definition

The Form Element
- The Form Element
  - The title Attribute....14
  - The style Attribute....15
    - normal style forms....15
    - dialog style forms....15
  - The name Attribute....15
  - The readonly Attribute....15
- The Options and Option Elements
- The Parameter Element
  - The name Attribute....16
  - The default attribute....16
  - The description Attribute....17
  - The required Attribute....17
  - The type attribute....17
  - Defining Parameters in the gfd....17
  - Setting Parameter Values During Startup....17
  - Learning More About Parameters....17

Datasources
- Table-Bound Datasources
- Static Datasources
- Defining Conditions

Linking Datasources via Master/Detail
Defining Master/Detail Datasources....19
Miscellaneous Settings
Pre-Query....20
Events and Triggers
Overview of Triggers
Trigger Types....21
Named Triggers....21
Object Triggers....21
The Trigger Namespace....21
Overview of Events
Event Levels....22
A Sample Table and Form
The Table Definition....22
The Form Definition....23
Your First Trigger – Validating Quantity Amounts....24
Form-Level Triggers
On-Startup....27
On-Activate....27
On-Exit....27
Pre-Commit....27
Post-Commit....27
Block-level Triggers
Pre-Query....28
Post-Query....28
Pre-Change....28
Pre-Insert....28
Pre-Update....28
Pre-Delete....29
Pre-Commit....29
Post-Commit....29
On-NewRecord....29
On-RecordLoaded....29
Pre-FocusIn....30
Post-FocusIn....30
Pre-FocusOut....30
Post-FocusOut....30
Working with Fields
Typecasting Fields
Default Values
Default (New Records)....31
Default (Queries)....31

Default to Last Entry.....31
Setting via Triggers.....32
Controlling User Input with Properties
Allow Editing.....32
Required.....32
Limiting Length of Text.....32
Trimming Spaces.....32
Controlling User Queries with Properties
Ignore Case on Query.....33
Allow Queries.....33
Designing for Multiple Interfaces
Trigger Recipes
Timestamping a Record prior to a Commit
Stamping a Record with User's Login prior to a Commit
Auto-Populating an Entry from a Sequence
Invoking gnue-forms
Command line
Symlinks

Chapter 1: Introduction

GNUe's Face to the World

Overview

What is GNUe Forms?

GNUe Forms provides an easy to use, consistent front end to your applications. Programmers are freed from the need to write the logic required to access data or define UI behavior. Training is reduced because personnel can expect a baseline of functionality across all GNUe Forms based applications.

What GNUe forms is not

GNUe Forms is not a program that allows one scan a paper form to fill out via computer. While it is easy to use GNUe Forms to produce a UI that can populate a pre-printed form, it will not display a graphical representation of the actual form on the screen for the end user to fill out.

Features of GNUe Forms

In addition to numerous features shared across all GNUe components, GNUe Forms provides the following features:

- Ultra rapid application development when coupled with GNUe Designer
- Input validation and transformation without writing a single line of code
- Complete support for data manipulation without writing a single line of code
- Automatic support for Master/Detail relationships in between data sources
- Embedded triggers written in python to extend the functionality of GNUe Forms
- Cross platform support across Macintosh, *nix, Windows, and text terminals
- Transparent network support for form storage

Key Concepts

Before designing an application for Forms, the developer should be somewhat familiar with a few key concepts:

- *Database Design* - This guide does not delve into database design. It is assumed the developer can either create his own tables, or has an existing set of tables to work with.
- *Python Scripting* - GNUe uses Python for scripting/event support. Any level of serious applications programming will likely require some level of Python. There is a short section entitled "A Brief Introduction to Python" in this guide that can serve as a starting point. The average programmer can likely learn enough simply by trying out the examples in this guide.
- *XML* - GNUe extensively uses XML for its internal storage format. While it is possible to create GNUe applications via Designer without interacting with the XML formats, a good, solid understanding of XML basics would definitely be useful.

Anatomy of a GNUe Form

Typically GNUe Forms reads the definition of a form from a human readable XML-based GNUe Form Definition (gfd) file. In some cases this gfd file is generated automatically at runtime by an application such as GNUe Appserver.

The GFD defines the sources of data that will be utilized, the relationships between the sources of data, the user interface to link to the data, and any validations or transformations that must be done during data entry.

Every GFD file can be thought of as consisting of three major components.

- The form itself which contains all form specific configuration details, such as the options, datasources, and parameters defined. It also contains the other two major components, which are logic and layout.
- The logic component which defines how the form will handle the information it contains.
- The layout section containing the defintion of the UI to present to the user.

When presented together the basic structure of the gfd would look as follows

Form
- **Options**
- **Parameter**
- **Datasources**
- **Logic**
  - **Blocks**
    - **Field**
- **Layout**
  - **Pages**
    - **Labels, Entries, Buttons, etc**

Without going into too much detail we'll step through the purpose of each of the various components of the form.

Options

Options provide the form designer with a method of specifying form specific settings to a form. Options would include things such as the version information for the form.

Parameters

Parameters allow the form designer to define arguments that can be passed into a form at runtime. Parameters also allow a form to pass values back and forth to a special type of dialog form. This is useful when a form designer needs to create custom dialogs that are called from the main form.

Datasources

Datasources are, exactly as their name implies, the sources of data that can be utilized in the retrieval and storage of information. If you are going to link to a table in an RDBMS, or populate a dropdown on the screen, or access information via a trigger, you will use a datasource.

Logic

The logic section of a form is a container used to hold the objects that define most of the behavior of the form. Here is where you will define blocks and the fields they contain and link them with datasources. It is worth noting that while the logic section can influence the user interface, such as forcing all input in a field to uppercase, the logic section is not the user interface.

Blocks

Blocks are responsible for storing, manipulating, and navigating any data associated with a datasource. They define how a form should interact with a datasource. They are also used to define some default settings for the fields they contain.

Fields

Fields provide a 1:1 link between a field of information in the block and a field of information in a datasource (think field in a table, or property of an appserver object). Fields are also where the form designer specifies validation and transformation steps that need to be observed. An example would be to limit the length of a customers first name to 30 characters, while forcing all input to uppercase regardless of what case was entered.

Layout

The layout section is responsible for the size, layout, and widgets that make up the user interface of the form.

Pages

Page sections act as containers for all widgets that will be displayed on the UI. They allow a form that would be too large to display all at once to be organized into screens or tabs.

Labels, Entries, Buttons

The components that comprise the user interface of the form. They work in a similar fashion to most modern widget sets with some subtle differences that will be explained later in this guide.

Triggers

While it is not apparent from outline above, GNUe Forms supports embedded chunks of code called triggers. Triggers allow the form designer to perform computations or add additional functionality to forms. Example would include printing forms, processing credit cards via an online gateway service, or driving a document scanner,

A Sample GNUe Form Definition

OK, let's take a look at our first complete form definition file. Our form will provide an interface used to maintain a table containing address information. The schema of this table is as follows:

```
Column    |          Type
----------+-----------------------
name      | character varying(50)
address1  | character varying(50)
address2  | character varying(50)
city      | character varying(30)
zip       | character varying(10)
```

If we were to pass this definition into GNUe Forms the result would look similar to the following

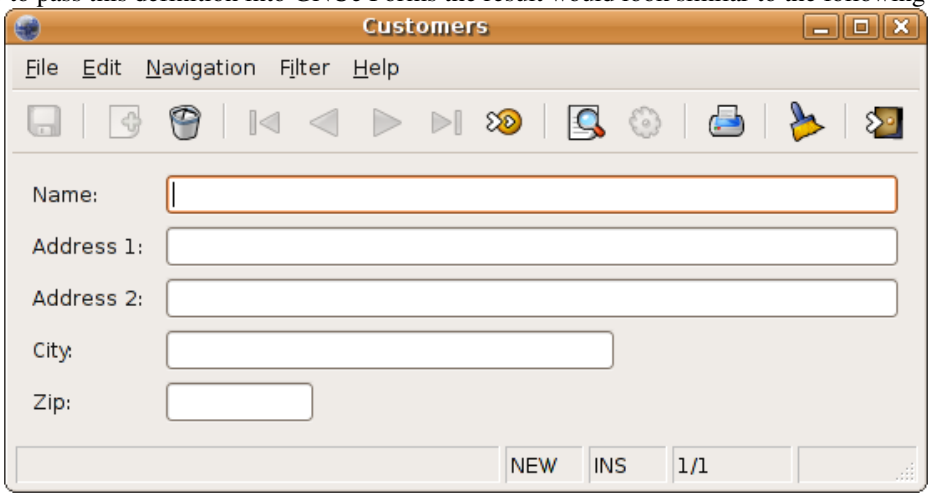

Figure 1: address.gfd as rendered by the wx26 driver

The code to define this form is as follows:

```
<?xml version="1.0"?>
<form title="Customers">
  <datasource name="dts_customer" connection="sample" table="customer">
    <sortorder>
      <sortfield name="name" ignorecase="Y"/>
      <sortfield name="city" ignorecase="Y"/>
    </sortorder>
  </datasource>
  <logic>
    <block name="blk_customer" datasource="dts_customer">
      <field name="name" field="name" datatype="text" length="50"/>
      <field name="address1" field="address1" datatype="text" length="50"/>
      <field name="address2" field="address2" datatype="text" length="50"/>
      <field name="city" field="city" datatype="text" length="30"/>
      <field name="zip" field="zip" datatype="text" length="10"/>
    </block>
  </logic>
  <layout>
    <page name="pg_customer" caption="Customer">
      <vbox name="box" block="blk_customer">
        <entry name="name" field="name" label="Name:"/>
        <entry name="address1" field="address1" label="Address 1:"/>
        <entry name="address2" field="address2" label="Address 2:"/>
        <entry name="city" field="city" label="City:"/>
        <entry name="zip" field="zip" label="Zip:"/>
      </vbox>
    </page>
  </layout>
</form>
```

Now to break the form apart line-by-line.

```
<?xml version="1.0"?>
```

The XML declaration as required by standard XML. This line is used to define the version of XML as well as the character encoding used in the document. While GNUe Forms doesn't require this line, you should include it to maintain compatibility with future versions of XML tools.

Next we start the form defintion.

```
<form title="Customers">
```

Nothing major here we've started the form and defined it's title as “Customers”. Most graphical environments will display this title on window borders and task bars.

Next we'll be setting up our datasource. Datasources establish a connection between our form and the source of the data the form will be manipulating.

```
  <datasource name="dts_customer" connection="sample" table="customer">
    <sortorder>
      <sortfield name="name" ignorecase="Y"/>
      <sortfield name="city" ignorecase="Y"/>
    </sortorder>
  </datasource>
```

Our datasource definition accomplishes the following steps:

- Creates a datasource named “dts_customer”.
- Instructs the datasource to establish communication with the connection named “sample” as

defined in the connections.conf file.

- Links the datasource to the table named “customer”. This is the only place the table name is referenced in a gfd, all gfd references requiring information from this table willreference the name of the datasource.
- Sets the default sort order for information pulled from the datasource.[1]

Now that we've configured the datasources we'll be using for this form it is time to move into the logic section. Which is accomplished by the following line.

```
<logic>
```

Now that we've entered the logic section of our form, it is time to setup a block.

```
    <block name="blk_customer" datasource="dts_customer">
```

This line defines a block named “blk_customer” which is associted to the dts_customer datasource we defined above. The fields we will soon define inside this block will have access to the fields of the table linked to the zips datasource.

Now it is time to add fields to our block. Fields define a relationship between a block and the fields of information stored in the datasource interfaced to the block. Fields also define any validation or transformation the form designer wishes to enforce upon the input data. So let's setup our first field.

```
        <field name="name" field="name" datatype="text" length="50"/>
```

We have now defined a field named “name” which has been linked to the “name” field in the “customer” table. There is no need to specify the dts_customer datasource as fields use the datasource associated with their containing block. The field accepts text up to 50 characters.

The other fields are added likewise:

```
        <field name="address1" field="address1" datatype="text" length="50"/>
        <field name="address2" field="address2" datatype="text" length="50"/>
        <field name="city" field="city" datatype="text" length="30"/>
        <field name="zip" field="zip" datatype="text" length="10"/>
```

We have now completed adding all the fields we require in this block so let's close it.

```
    </block>
```

As our simple example only requires the single block we can now close the logic section as well.

```
</logic>
```

Now that we've completed the logic section of our form it is time to define how the form will be presented to the user. This is accomplished in the layout section of a form definition which we open next.

```
<layout>
```

We start by creating a page to place widgets upon.

```
    <page name="pg_customer" caption="Customer">
```

Since this form is only one page long the caption is not really important. Later, when we are using tabbed forms the caption is what will be displayed on the tab.

Next, we define how we want the widgets arranged on this page of the form. We want them arranged vertically, so we use a vertical box (vbox):

---

1 this sort order does not affect the order of records entered via the form. Newly added records will be displayed in the order in which they are entered.

```
      <vbox name="box" block="blk_customer">
```

This definition also defines the block this box is bound to. This does not only mean that all entries will be attached to fields of the block “blk_customer” (unless explicitly stated otherwise), it also tells graphical user interfaces to make the mouse wheel navigate through this block if the mouse pointer is within the box. Note that since the box has no label attribute, it will not display a visible border in the user interface.

Now that the box is defined we can place the entries in it:

```
        <entry name="name" field="name" label="Name:"/>
```

The entry is bound to the field “name” of the block “blk_customer” (remember we defined the block in the enclosing vbox) and is labelled “Name:” on the screen. Everything else, like position on the form, or size of the entry is decided automatically from the information given in the field definition.

We add the rest of the entries likewise:

```
        <entry name="address1" field="address1" label="Address 1:"/>
        <entry name="address2" field="address2" label="Address 2:"/>
        <entry name="city" field="city" label="City:"/>
        <entry name="zip" field="zip" label="Zip:"/>
```

Since we have not specified an entry style, which we will cover later, the entries default to standard input.

Note GNUe Forms handles input widgets differently than many other applications. All input is handled via an entry object to which you apply a style. So instead of specifying a type of widget such as a checkbox, dropdown, or text field, request an entry of style checkbox. You may asking yourself what is the difference between specifying an entry with a style of checkbox vs specifying a checkbox directly. The difference is that GNUe Forms is designed to use the same core logic across numerous front end interfaces; and to fail gracefully to a functional state even when a form is ran on an interface which is incapable of displaying the requested style.
For example, lets assume that we have an entry of style dropdown which would display properly on most modern GUIs. However we are suddenly faced with running the form on some old VT style text terminals where an older copy of GNUe Forms is in use and doesn't have any concept a dropdown. In these cases GNUe Forms simply ignores the style request and presents a standard entry. While the form may not possess it's full functionality it would still be usable.

Tip By separating the block field from the layout entry it is possible for the form designer to create multiple entries across multiple pages of a form which all point back to the same block field. Every entry is fully functional, and since validation and transformation is done at the block field level there is no risk of multiple entries allowing invalid information into the backend. Entry styles, which are covered later in this guide, allow even greater flexibility when using multiple entries per block field.

Finally we have completed our form. All that remains is to close the vbox, the page, the layout section, and finally the form.

```
      </vbox>
    </page>
  </layout>
</form>
```

Chapter 2: The Form Element

Configuring the form wrapper

As a parent container of a form the form element defines the various configuration options, the logic, and layout sections of a form. This chapter examines the form element and the features it makes available to the form designer.

The Form Element

All gfd files contain at least one <form> element. The basic structure of a form element is as follows.

<form>

Function:

Top-level element that encloses all the logic and visuals that the user interface will show to the user.

Attributes:

NAME	STYLE
READONLY	TITLE

Contains Elements:

action	import-trigger
connection	layout
datasource	logic
dialog	menu
import-action	options
import-datasource	parameter
import-dialog	toolbar
import-layout	trigger
import-logic	

The title Attribute

The optional title attribute is used to specify a title the form designer would like to see displayed on the window decoration and/or taskbar.

The style Attribute

A form element may be one of two styles: normal or dialog.

normal style forms

A form of normal style is the default and is typically what most form designers will use for most of their work.

dialog style forms

Occasionally however a form designer may wish to display a custom dialog box to provide the end user with more information or prompt for additional information or confirmation. When such a dialog is required then a additional form definition will be placed into the gfd with a style of dialog.

Dialog style forms match feature for feature anything you can do inside a regular form. The only difference

is that they are modal in nature.

Note GNUe Forms supports a tag <dialog> which is simply an alias for <form style="dialog">

Caution A form containing a dialog style form in addition to a normal form cannot be edited in GNUe Designer. GNUe Designer currently cannot handle the concept of more than one form in a single gfd and will corrupt your data.
Currently the easiest way to alter gfd containing more than one form definition is as follows

1. Make a copy of the original form definition file.
2. Using your favorite text editor. Remove all form elements from the copy with the exception of the one you wish to alter.
3. Use GNUe Designer to make the desired alterations in the copy. Save your work.
4. Using your favorite text editor. Replace the form element in the original file with the form element from the copy.

The name Attribute

In typical form definition the name attribute can be considered optional. The only time a name is required would be when a gfd contains a primary form and one or more forms of style dialog. The name will then be used in triggers to request display of the dialog form.

The readonly Attribute

When the optional readonly attribute is set to Y the form will no longer allow new records or modifications to existing records to take place.

By utilizing this attribute it is possible to provide forms that permit users to lookup information they would otherwise be able to alter. An example of this might include access to customer information outside a companies main customer management system. Though a user account may typically be able to alter information inside the primary system it has proven cumbersome to query information using the same system. A gfd could provide a quick and easy interface to search for information while preventing the user from making changes that would bypass the primary system.

The Options and Option Elements

The form element, as well as several other elements, inside a gfd can contain an options element. In the case of the forms element the options allow the form designer to provide a subset of the Dublin Core Metadata stored inside option elements. A sample options section would appear as follows inside a gfd.

```
<form title="My form">
  <options>
    <option name="version" value="0.0.1"/>
    <option name="author" value="John Doe"/>
  </options>
  The rest of the form defintion.....
</form>
```

Option elements have two attributes; name and value. Option names currently supported at the form level include author, title, and version.

Note The title option can be used in place of the title attribute assigned to a form element. They are functionally equivalent.

The Parameter Element

Parameter elements allow the form designer to specify runtime parameters that can be utilized by various components in a form. Parameters allow for the following

- Setting query condition values for datasources
- Communicating between a form and its dialog forms.
- Setting initial values from the command line when opening a form.

The structure of a parameter is as follows:

<parameter>

Function:

A form can get parameters from the outer world or a calling form, and can pass values back too in these parameters.

Attributes:

NAME	LENGTH
DATATYPE	REQUIRED
DEFAULT	SCALE
DESCRIPTION	

Contains Elements:

None

The name Attribute

This attribute defines the unique name of the parameter which will be referenced in other parts of the gfd.

The datatype attribute

Specifies the data type of parameter. Defaults to text. Possible values include text, number, date, time, datetime, and boolean.

The default attribute

The required attribute sets the default value of the parameter if no value is specified during the activation of the form.

The description Attribute

This optional attribute can store a short description of the purpose of the parameter. In practice it is not currently being used, but it might at some point be used for a help text.

The length Attribute

For parameters of datatype text and number, this attribute contains the maximum length allowed for this parameter.

The required Attribute

If set to Y, this attribute flags GNUe Forms that an input value must be provided during form activation.

The scale attribute

For parameters of datatype number, this attribute defines the scale. For example, a length of 6 and a scale of 2 means a numerical format of 9999.99.

Defining Parameters in the gfd

Parameter elements appear immediately inside the form element, they are not nested inside any other elements. To define a set of parameters named foo and bar the following code would be used:

```
<form>
  <parameter name="foo" default="abc"/>
  <parameter name="bar" default="xyz"/>

  The rest of the form definition...

</form>
```

Setting Parameter Values During Startup

Once defined, parameters can be passed into a form during startup via the command line. If, for exmaple, we wished to override the default foo value we defined above we would execute GNUe Forms as follows:

```
gnue-forms myform.gfd foo=lmnop
```

Learning More About Parameters

Parameters are typically used by datasources and inside triggers. Please see their respective chapters for additional information

Chapter 3: Datasources

A Datasource links data to our form. Usually, a datasource points to a table if using a relational database, or a appserver class if using GNUe AppServer. A form can have several datasources if pulling data from multiple locations, or no datasources at all if the form does not reference outside data.
If a form does not have a datasource, a *virtual datasource* is created. The commit, rollback, and query functions do not serve a purpose against virtual datasources. This is particularly useful for action forms that simply cause actions to occur, but do not directly manipulate data.
Datasources can be linked to each other in a master/detail fashion via a foreign key. In essence, each time the master datasource changes, the detail datasource is automatically requeried to bring up records related to the master. See the section on master/detail relationships for more information.

<datasource>

Function:

A form can get parameters from the outer world or a calling form, and can pass values back too in these parameters.

Attributes:

NAME	ROWID
TYPE	EXPLICITFIELDS
for type "object":	*for type "sql":*
CONNECTION	CONNECTION
TABLE	*for all types:*
CACHE	MASTER
DISTINCT	MASTERLINK
REQUERY	DETAILLINK
PRIMARYKEY	PREQUERY

Contains Elements:

for type "object":	*for type "sql":*
condition	sql
sortorder	*for type "static":*
	staticset

GNUe Forms supports three different types of datasources, related to where the data is read from:

Datasources of type "object"

These datasources are the most common ones. They are connected to a table defined by the "table" attribute through the connection given in the "connection" attribute.
Several other attributes are available for special cases:

The cache Attribute

This defines the number of records to fetch from the backend at a time. For datasources where you expect a higher number of records, setting this to a high number might increase the performance, because it tells GNUe to retrieve data from the backend in bigger chunks.

The distinct Attribute

If this is set to “Y”, this will cause GNUe to remove duplicates from the result of any query done to this datasource. Readers with SQL knowledge might notice this is the same behaviour as including a “DISTINCT” in the SELECT command.
Datasources with the “distinct” attribute set are not writable.

The requery Attribute

GNUe is always prepared to react on server side triggers on the database backends. By default, after saving any changes to the database server or the GNUe AppServer, it will requery the records it has just saved to see if the backend did any automatic changes to the data.
If you don't use server side triggers, this is not necessary and generates additional load on the network and the backend, and you can turn it off by setting the requery attribute to “N” explicitly.

The primarykey Attribute

For writing changes back to the backend, GNUe Forms has to identify the record it wants to modify. For this, it uses the following methods:

1. Some backends use a predefined field name for the primary key. Most notably, this is GNUe AppServer's gnue_id field.
2. For other backends, the user can define a primary key through the primarykey attribute.
3. Some SQL database backends have an “oid” or “rowid” attribute on every table. GNUe Forms uses this if no primary key is defined.
4. If neither a primary key nor a oid/rowid is defined, GNUe Forms uses all fields used in the form to identify the record. This is an insecure method, and it is always recommended to define a primary key if the form should run on backends that don't support oid/rowids.

The rowid Attribute

With this attribute, the form designer can hint GNUe forms to a rowid field supported by the database but unknown by the database driver. This should be needed only in very rare cases.

The explicitfields Attribute

Here you can list field names of database fields that should be queried additionally to the fields defined in the block attached to this datasource. You don't need this unless you do some very hackish things that you shouldn't do anyway.

Datasources of type “sql”

With these datasources, you can send direct SQL queries to the database backend defined through the “connection” attribute. Although it is a very powerful possibility, it is only useful against SQL database backends.
Direct SQL datasources are not writable.

Static Datasources

Static datasources are not bound to an external source of data, but represent a predefined and constant set of data.

```
<datasource name="AvailDS" type="static">
  <staticset fields="id,descr">
    <staticsetrow>
      <staticsetfield name="id" value="A"/>
      <staticsetfield name="descr" value="Available"/>
    </staticsetrow>
    <staticsetrow>
      <staticsetfield name="id" value="N"/>
      <staticsetfield name="descr" value="Not Available"/>
    </staticsetrow>
    <staticsetrow>
      <staticsetfield name="id" value="B"/>
      <staticsetfield name="descr" value="Backordered"/>
    </staticsetrow>
  </staticset>
</datasource>
```

Static datasources ar often used for lookup sources to map predefined codes with the user visible explanations.

Defining Conditions

Forms' datasources of type “object” support conditions. Conditions place restrictions on the records returned by a datasource. For those familiar with SQL, a condition translates directly into a `WHERE` clause.

```
<datasource connection="test" table="reps">
  <condition>
    <or>
      <eq>
        <cfield name="active"/>
        <cconst value="Y"/>
      </eq>
      <gt>
        <cfield name="sales_ytd"/>
        <cconst value="0"/>
      </gt>
    </or>
  </condition>
</datasource>
```

In this example, we are basically only allowing records from the `reps` table where the representative is either active (`active = 'Y'`) or had sales this year.

“cfield” is a database field used in a condition, “cconst” a constant value to compare with. “cparam” can be used for comparison with the current value of a parameter.
Available operations in a condition tree are:

- logical operators: “and”, “or”, “not”
- mathematical operators: “add”, “sub”, “mul”, “div”, “negate”
- comparison operators: “eq”, “ne”, “gt”, “ge”, “lt”, “le”, “like”, “notlike”, “between”, “notbetween”, “null”, “notnull”
- string operators: “upper”, “lower”

- the “exist” operator.

The “exist”operator can be used like in this example:

```
<datasource name="dtsCustomer" connection="gnue" table="customer">
  <condition>
    <exist table="invoice" masterlink="id" detaillink="customer">
      <ne>
        <cfield name="paidamount"/>
        <cfield name="totalamount"/>
      </ne>
    </exist>
  </condition>
<datasource>
```

This condition will only return those customers that have at least one invoice that isn't fully paid.

Defining Sort Order

To define the sort order for datasources of type “object”, you can use the sortorder and sortfield elements:

```
    <sortorder>
      <sortfield name="name" ignorecase="Y"/>
      <sortfield name="city" ignorecase="Y"/>
    </sortorder>
```

The sortfield element has the following attributes:

- name: field name in the database
- descending: if set to “Y”, the sorting order is reversed
- ignorecase: if set to “Y”, the case is not regarded in sorting for alphanumeric fields.

Linking Datasources via Master/Detail

Quite often, you will want a second datasource's behavior to be tied to a primary datasource. If a new record is queried in the first datasource, all corresponding records in the second datasource should automatically appear. Likewise, creating a new record in the first datasource should clear out the second datasource and any subsequent new records in this second datasource will be automatically associated with the new primary record.
In GNUe, we call this relationship a Master/Detail relationship. It closely mirrors the concept of primary/foreign keys in relational databases.
Master/detail relationships have the following properties:

- When a new record is created in the master datasource, the detail datasource will have no records.
- When the master source is queried, a new set of records is queried in the child datasource for each record in the master source. This happens on an as-needed basis so as not to waste resources.
- On posting, or saving, a detail record, its “foreign key” is automatically populated with the “primary key” of the master record.

Defining Master/Detail Datasources

There are no special attributes for a master datasource to indicate its role as master.
The detail datasource, on the other hand, has three special attributes that must be provided: `master`, `masterlink`, and `detaillink`.

- `master:` This should be the name of the master datasource.
- `masterlink`: This is the name of the field or fields in the master datasource that links it to the child datasource. If there are multiple fields, i.e., the master primary key is “composite”, then `masterlink` should be a comma-separated list of field names.
- `detaillink`: This is the name of the field or fields in the detail datasource that links it to the child datasource. If there are multiple fields, i.e., the master primary key is “composite”, then `detaillink` should be a comma-separated list of field names with the order of the fields corresponding to the order provided in `masterlink`.

Miscellaneous Settings

The prequery attribute

Datasources support a prequery attribute. If set to true, then on form startup, the datasource is populated and the data initially displayed on screen. If set to false (the default), then the form starts out with an empty record until either the user or a trigger queries the database.

Chapter 4: Events and Triggers

Customizing GNUe Forms Behaviour

In this chapter we will be covering events and triggers. Triggers allow a form designer to define custom code inside a gfd. These triggers can then be associated with form events which “fire” the trigger whenever the specified event occurs. Triggers and events are tightly intertwined and it is not uncommon for people to mix the terms when discussing form design.

Overview of Triggers

Triggers are custom code written in python allow a form designer to extend the capabilities of their gfds. With triggers you can perform actions such as

- manipulate data within a form
- interface with external applications and hardware
- perform data manipulation on datasources that are not part of the form's layout

Trigger Types

Triggers can be one of two types, named or object.

Named Triggers

A named trigger can almost be thought of as a function defined within a GFD. It is attached directly to the form element and can be used as the source for an object trigger, or can be called from within an object trigger or from another named trigger. Named triggers are useful in situations where you have computed field which is dependent on the value of several other form fields. By creating a single named trigger to compute the single field value, and attaching it to each of the source fields you can reduce the amount of code you have to maintain.

Object Triggers

Object triggers attach directly to objects such as a form, a block, or a field and are tied to an event associated with that object. The object to which the trigger is attached greatly effects how the trigger is processed. For example by attaching a trigger to a change event against a field the trigger would fire any time that field's value changed. A trigger attached to a charge event against a block would fire any time any field in that block had it's value changed.

The Trigger Namespace

A namespace, for those unfamiliar with the term, can be thought of as a container which defines the variables available to the form designer. All GNUe Forms triggers run within a namespace built from the name attributes you assign elements inside the gfd, as well as some reserved variables GNUe Forms builds from the system wide installation and configuration information.

The purpose of this namespace is to shield the form developer from the GNUe internals while providing them the ability to manipulate the form and runtime environment programmatically. Since namespace usage is integral to utilizing triggers, you will be exposed to it as we cover the usage of events and triggers.

Overview of Events

GNUe Forms generates an event whenever specific actions occur within a form. One of more triggers can be attached to an event so that the custom code defined in the trigger will be executed during the event.

Event Levels

If you recall the anatomy of the gfd as described earlier in this guide you will note it looks like an inverted tree. Events are defined at several object levels in the gfd. The level of the event dictates when triggers attached to the event will execute. Events occur at the form, block, field, and entry object levels.

A Sample Table and Form

In order to make the examples in this chapter easier we will be using a simple table and form setup. In our fictional example we will be creating a table and form to track items in an inventory.

The Table Definition

```
                  Table "public.inventory"
    Column     |         Type          | Modifiers
---------------+-----------------------+-----------
 item_id       | numeric               | not null
 replacement_id | numeric               |
 description   | character varying(20) |
 qty_on_hand   | numeric               |
 qty_on_order  | numeric               |
 qty_reserved  | numeric               |
Indexes:
    "inventory_pkey" PRIMARY KEY, btree (item_id)
Foreign-key constraints:
    "inventory_replacement_id_fkey" FOREIGN KEY (replacement_id)
                                    REFERENCES inventory(item_id)
```

Here we've defined a table that contains an item with a description and three inventory quantities, the number of the item we have on hand in our warehouse, the number of items currently on order from our vendors, and the number of items reserved for existing orders. Additionally we provide an id that points to another item in the same table that acts as a replacement item if/when our current item is discontinued.

Note For those of you reading this guide that find themselves cringing at such horrific table design. Please accept my apologies, however please remember this guide is to teach concepts of GNUe, not good table design. Thus this example is attempting to minimize complexity in the sample table so we can concentrate on accomplishing goals in GNUe. Having said that, I truly am sorry :)

The Form Definition

Initially, our sample form will consist of a set of fields in a grid layout. During the course of this chapter you will be utilizing triggers to extending the base functionality of the form.

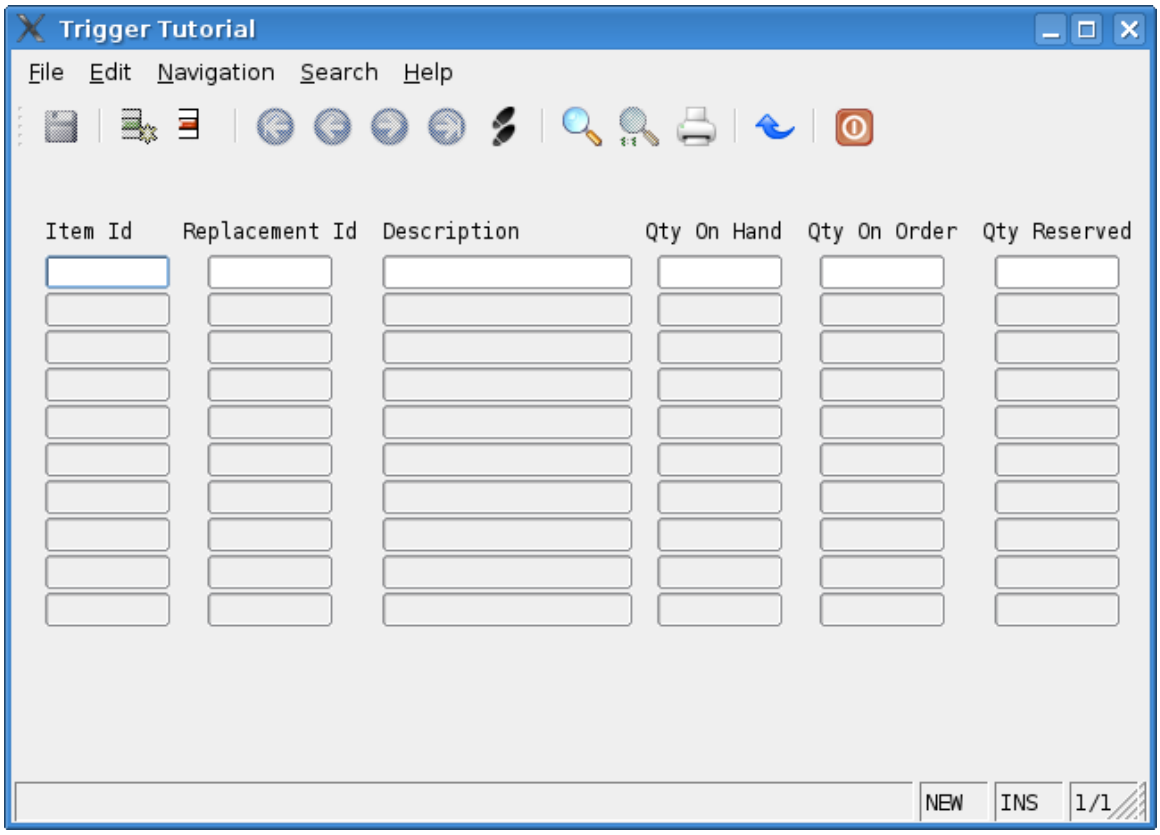

Figure 2: The initial trigger tutorial form

The code to define our form is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>

<form title="Trigger Tutorial">
  <options/>
  <datasource name="dtsINVENTORY0" connection="gnue" table="inventory"/>
  <logic>
    <block name="blkINVENTORY" datasource="dtsinventory0" rows="10">
      <field name="fldItemId" field="item_id" datatype="number"/>
      <field name="fldReplacementId" field="replacement_id" datatype="number"/>
      <field name="fldDescription" field="description" length="20"/>
      <field name="fldQtyOnHand" field="qty_on_hand" datatype="number"/>
      <field name="fldQtyOnOrder" field="qty_on_order" datatype="number"/>
      <field name="fldQtyReserved" field="qty_reserved" datatype="number"/>
    </block>
  </logic>

  <layout xmlns:Char="GNUe:Layout:Char" Char:height="13" Char:width="89">
    <page name="Page1">
      <label name="lblQtyReserved" Char:height="1" Char:width="7" Char:x="1"
             Char:y="1" text="Item Id"/>
      <entry name="entItemId" Char:height="1" Char:width="10" Char:x="1"
             Char:y="2" block="blkINVENTORY" field="fldItemId" label=""/>
      <label name="lblQtyReserved" Char:height="1" Char:width="14" Char:x="12"
             Char:y="1" text="Replacement Id"/>
      <entry name="entReplacementId" Char:height="1" Char:width="10"
             Char:x="14" Char:y="2" block="blkINVENTORY"
             field="fldReplacementId" label=""/>
      <label name="lblQtyReserved" Char:height="1" Char:width="11" Char:x="28"
             Char:y="1" text="Description"/>
      <entry name="entDescription" Char:height="1" Char:width="20" Char:x="28"
             Char:y="2" block="blkINVENTORY" field="fldDescription" label=""/>
      <label name="lblQtyReserved" Char:height="1" Char:width="11" Char:x="49"
             Char:y="1" text="Qty On Hand"/>
      <entry name="entQtyOnHand" Char:height="1" Char:width="10" Char:x="50"
             Char:y="2" block="blkINVENTORY" field="fldQtyOnHand" label=""/>
      <label name="lblQtyReserved" Char:height="1" Char:width="12" Char:x="62"
             Char:y="1" text="Qty On Order"/>
      <entry name="entQtyOnOrder" Char:height="1" Char:width="10" Char:x="63"
             Char:y="2" block="blkINVENTORY" field="fldQtyOnOrder" label=""/>
      <label name="lblQtyReserved" Char:height="1" Char:width="12" Char:x="76"
             Char:y="1" text="Qty Reserved"/>
      <entry name="entQtyReserved" Char:height="1" Char:width="10" Char:x="77"
             Char:y="2" block="blkINVENTORY" field="fldQtyReserved" label=""/>
    </page>
  </layout>
</form>
```

This example still uses the old (deprecated) character position based layout. However, the possibilities of the triggers are still the same.

Your First Trigger – Validating Quantity Amounts

Though this form is relatively simple, it already permits the user to enter, update, and modify information stored in our inventory table. Additionally the form prevents the end user from entering letters into quantity fields. You are now going to add the following additional functionality

- Block negative quantities since it is not possible to have an on hand quantity of -5 sitting on the

shelf.

- Because our fictional inventory rarely has more than 99 of any item in stock we will issue a warning whenever an on-hand quantity entered is greater than 99.

To do this we will be adding a trigger to our field named fldQtyOnHand. Find the line in the gfd that reads

```
<field name="fldQtyOnHand" field="qty_on_hand" datatype="number"/>
```

And replace it with the following

```
<field name="fldQtyOnHand" field="qty_on_hand" datatype="number">
  <trigger name="NoNegOnHand" type="PRE-FOCUSOUT"><![CDATA[
    value = self.value

    if value > 99:
        show_message("Quantity over 99 entered.  Please verify.")
    elif value < 0:
        abort("Negative quantity entered")
  ]]></trigger>
</field>
```

Now we have embedded a trigger named “NoNegOnHand” inside the field. You will notice that the trigger code is written in pure python. Lets look at these changes line by line.

```
<field name="fldQtyOnHand" field="qty_on_hand" datatype="number">
```

This line is the exact same as the original line with one exception. The backslash (/) is missing at the end. The XML standard requires all entities such as our <field> always have an opening tag and a closing tag which has the same name but is preceded by a backslash (/). So in the case of our field enties the opening tag is <field> and the closing tag is </field>. This can add quite a bit of text to an XML file so the standard provides for a way to combine the tags into one. Entities that do not contain any additional enties can add a backslash at their end to close out the tag. Now that we are embedding a trigger inside the field we had to remove that blackslash.
Now it is time to define the trigger

```
<trigger name="NoNegOnHand" type="PRE-FOCUSOUT"><![CDATA[
```

We have defined a trigger named NoNegOnHand which is embedded inside our field. The code **type="PRE-FOCUSOUT"** configures this trigger to fire (execute) whenever it's container object, in this case the field, emits a PRE-FOCUSOUT event.

Finally the text **<![CDATA[** is used to start an XML CDATA section. Trigger text often contains symbols such as “>” and “<” which confuse XML parsers. When the parser encounters a symbol such as “<” it assumes that it is at the start of a new element. CDATA sections tell the XML parser “do not parse the text inside as XML”, allowing your code to not have to result to using special characters like **>** every place a “>” is needed.

Now it is time to begin the trigger's program code. First we'll grab pull a copy of the current value of the field.

```
value = self.value
```

You will notice that instead of referencing the field by it's name we have used the special namespace variable **self**. Inside trigger code, **self** is a reference to element containing the trigger defintion. In this case the trigger is located inside the fldQtyOnHand field element, so **self** is a reference to that specific field.

Now it is time to perform the validation.

```
        if value > 99:
            show_message("Quantity over 99 entered.  Please verify.")
```

This standard python statement calls the built-in **show_message** function if the value of the field exceeds 99. **show_message** causes GNUe Forms to present the end user with the following dialog box:

Figure 3: The show_message dialog box

After the user selects OK to clear the dialog, the cursor is moved to the "Qty on Order" column, and processing of the form continues as normal. This allows the form designer to present the user with important information as they work.

Note You may have noticed that the comparison above uses the value 99 instead of the python string '99'. GNUe Forms always attempts to cast the value of fields to their datatype setting.

```
        elif value < 0:
            abort("Negative quantity entered")
```

Now we are validating that the value is not less than 0. If it is we call the built in **abort** method. The **abort** method causes two events to occur. First, the user is presented with a dialog box similar to the one generated by the showMessage method.

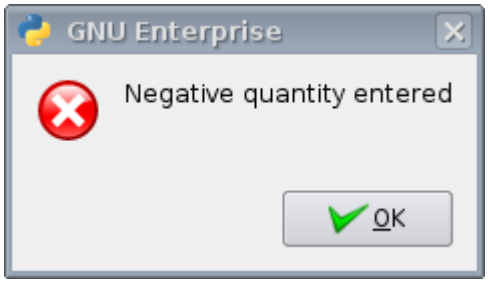

Figure 4: The alert dialog box

Second, the trigger aborts the event to which it is tied, which in this case is the PRE-FOCUSOUT event. While different types of events can react differently when aborted, the most common reaction is to return the form to the state it was in prior to the event. Thus in this instance the focus change was aborted. After the user selects OK they find the cursor has remained in the "Qty on Hand" column and will do so until they enter a non-negative value for the field.

So now we have all the code needed to warn about large quantities and prevent negative quantities from being enterer. The only thing left is to close the CDATA section, the trigger element, and finally the field element. Which is accomplished by

```
        ]]></trigger>
    </field>
```

Form-Level Triggers

A Form-level trigger is defined as an object that is activated at the form-level and is defined as a child of the form object.
The following form level triggers are defined:

On-Startup

The `On-Startup` trigger is executed once during the lifetime of a Form's instance.
Suggested uses for `On-Startup`:

- Setting initial flags
- Initializing global variables
- Setting parameters used to populate the blocks at form startup

Note that `On-Startup` runs before the blocks are populated with data, so it is not possible to set field values in a trigger bound to this event.
On-Startup triggers cannot abort the operation.

On-Activate

The On-Activate trigger is executed each time a form or dialog is activated.
Suggested used for `On-Activate`:

- Initializing field values, especially for unbound fields (fields not bound to a datasource)
- Moving the cursor to a specific entry

On-Activate triggers cannot abort the operation.

On-Statuschange

The On-Statuschange trigger is executed each time the status of the current block changes.
Suggested uses for On-Statuschange:

- Enable/Disable actions that are only useful depending on a specific state of the data (e.g. only for new records)

On-Statuschange triggers cannot abort the operation.

Pre-Exit

The Pre-Exit trigger is executed whenever the user tries to close the form in any way. It runs *before* the user is asked whether to save changes or not.
Suggested uses for Pre-Exit:

- Check whether the user is allowed to close the form.
- Automatically save changes on form close without asking the user.

Pre-Exit triggers can abort the closing of the form by calling the abort() function or raising any exception.

On-Exit

The `On-Exit` trigger is executed when either the user or a trigger requests that a form closes. It is called after Pre-Exit and after the user has been asked whether to save his changes. At the time On-Exit runs, there is no way back, it is sure that the form will close.
Suggested uses for `On-Exit`:

- Final clean-up

Pre-Commit

The `Pre-Commit` trigger is executed before a form-level commit occurs.
Suggested uses for `Pre-Commit`:

- Perform form-level validation of data

The Pre-Commit trigger can abort the operation.

Post-Commit

The `Post-Commit` trigger is executed after a form-level commit occurs.
Suggested uses for `Post-Commit`:

- Prepare form for next data to be entered.

The Post-Commit trigger cannot abort the operation.

Block-level Triggers

Most (but not all) block-level triggers are executed on a per-record basis. That is, a trigger would get executed once for every applicable record, not just once for the entire block.

Pre-Query

The `Pre-Query` trigger is executed when a user requests a query against the database by choosing the "apply filter" function. This trigger is unique from all other triggers in that it is called while the form is in *filter* mode. This means that any field changes made by this trigger don't actually modify a record, but instead are used as filter conditionals.
Suggested uses for a block-level `Pre-Query` trigger:

- Adding custom conditions to a query that are more complex than can be represented by a `field`'s `queryDefault` property.

Note that this trigger is *not* run when the query is initiated using the set_filter() method of the block programmatically.
The Pre-Query trigger can abort the query.

Post-Query

The `Post-Query` trigger is executed after a query against the database has been executed. A block-level `Post-Query` is executed once for each query.
Suggested uses for a block-level `Post-Query` trigger:

- Jump to a special record of the result set (instead of the first record)
- Calculate information from the complete result set, like a sum over all records.

Note that this trigger is *not* run when the query is initiated using the set_filter() method of the block programmatically.
The Pre-Query trigger cannot abort the query.

Pre-Change

The `Pre-Change` trigger is executed each time before any field in this block is modified. At the time the Pre-Change trigger is called, the modified field will still contain the old value.
The Pre-Change trigger cannot abort the change.

Post-Change

The Post-Change trigger is executed each time after any field in this block is modified. At the time the Post-Change trigger is called, the modified field already contains the new value.
Post-Change triggers cannot abort the change.

Pre-Commit

The Pre-Commit trigger is executed once for each block of the form before the changes are saved.

Pre-Insert

The `Pre-Insert` trigger is executed before a commit occurs on a record that is pending an insertion. A block-level `Pre-Insert` is executed once for each inserted record and is fired prior to the `Pre-Commit` trigger.
Suggested uses for a block-level `Pre-Insert` trigger:

- Stamping records with a creation date or created-by value
- Setting a primary key's value
- Setting other hidden, but pertinent, fields with default or pre-calculated values
- Storing historical information in transaction tables

Pre-Update

The `Pre-Update` trigger is executed before a commit occurs on a record that is pending an update. A new or deleted record is not considered "updated" for the purpose of this trigger. A block-level `Pre-Commit` is executed once for each changed record and is fired prior to the `Pre-Modify` trigger.
Suggested uses for a block-level `Pre-Update` trigger:

- Stamping records with a modification date or modified-by value
- Setting hidden, but pertinent, fields with default or pre-calculated values
- Storing historical information in transaction tables

Pre-Delete

The `Pre-Delete` trigger is executed before a commit occurs on a record that is pending a deletion. A block-level `Pre-Delete` is executed once for each record that has pending deletion. A block-level `Pre-Delete` fires prior to a `Pre-Commit` trigger.
Suggested uses for a block-level `Pre-Delete` trigger:

- Storing historical information in transaction tables

Pre-Commit

The `Pre-Commit` trigger is executed before a commit occurs on a record. A block-level `Pre-Commit` is executed once for each record that has pending changes, including new and deleted records. A block-level `Pre-Commit` fires prior to a Field's `Pre-Commit` trigger, but after the `Pre-Insert`, `Pre-Update`, and `Pre-Delete` triggers.
Suggested uses for a block-level `Pre-Commit` trigger:

- Stamping modified records with a date or modified-by value
- Setting hidden, but pertinent, fields with default or pre-calculated values
- Storing historical information in transaction tables

Note: If you noticed that Pre-Commit is used for two different purposes, you are right. This is a bug.

Post-Commit

The `Post-Commit` trigger is executed after a commit occurs. A block-level `Post-Commit` is executed once for each commit.

On-NewRecord

The `On-NewRecord` trigger is executed when a record is initially created. This trigger is executed once for each new record at the time of creation in the form.
Suggested uses for a block-level `On-NewRecord` trigger:

- Setting default values

On-RecordLoaded

The `On-RecordLoaded` trigger is executed after a record has been loaded from the database. A block-level `On-RecordLoaded` is executed once for each record that was loaded from a database. `On-RecordLoaded` is the counter-part to `On-NewRecord` in that one or the other should be executed for every displayed record.

Note `On-RecordLoaded` is only fired as a record is loaded from the database. This implies that, with GNUe's caching system, if only 10 records are displayed on screen at a time out of a table of 100 records, then only the first 10 or so records will have `On-RecordLoaded` fired. It is guaranteed, however, that by the time a the user can see a loaded record or before another trigger can be fired against a loaded record, `On-RecordLoaded` has already been called. It is possible to get around this by, at some point (for example in the `Post-Query` trigger), calling `block.lastRecord()` and (optionally, if needed) `block.firstRecord()`.

Suggested uses for a block-level `On-RecordLoaded` trigger:

- Populating non-database (automaticall calculated) fields.
- Resetting user-defined flags

Pre-FocusIn

The `Pre-FocusIn` trigger is executed as a new record is focused in a block. It is recommended that unless you have a specified understanding of the intention of forms, use `Post-FocusIn` instead of `Pre-FocusIn` as the latter trigger's behavior may change at some point to better reflect record focus.

Post-FocusIn

The `Post-FocusIn` trigger is executed as a new record is focused in a block. This may be triggered by a user navigating to a different record, by creating a new record, or by the user changing the focus in the form from one block to another one. The actual record change has occurred when this trigger is fired.
Suggested uses for a block-level `Post-FocusIn` trigger:

- Initialization of fields in some cases where the initialization value depends on what was entered before moving to this block.

Pre-FocusOut

The `Pre-FocusOut` trigger is executed on attempt to move the focus out of a record of this block. This may be triggered by a user navigating to a different record, by creating a new record, by attempting to save

the changes, by closing the form, by changing to filter mode, or by moving the UI focus out of the block. The actual record change has not occurred when this trigger is fired. It does not matter whether the operation is performed by the user or programmatically.
Suggested uses for a block-level `Pre-FocusOut` trigger:

- All kinds of block level input validation. It is guaranteed that there is no chance to save data from a block that has not run through the Pre-FocusOut trigger.

Post-FocusOut

The `Post-FocusOut` trigger is executed as a new record is focused in a block. It is recommended that unless you have a specified understanding of the internals of forms, use `Pre-FocusOut` instead of `Post-FocusOut` as the latter trigger's behavior may change at some point to better reflect record focus.

Field-level Triggers

Most (but not all) field-level triggers are executed on a per-record basis. That is, a trigger would get executed once for every applicable record, not just once for the entire operation.

Pre-Query

The `Pre-Query` trigger is executed when a user requests a query against the database by choosing the "apply filter" function. This trigger is unique from all other triggers in that it is called while the form is in *filter* mode. This means that any field changes made by this trigger don't actually modify a record, but instead are used as filter conditionals.
Suggested uses for a block-level `Pre-Query` trigger:

- Adding custom conditions to a query that are more complex than can be represented by a `field`'s `queryDefault` property.

Note that this trigger is *not* run when the query is initiated using the set_filter() method of the block programmatically.
The Pre-Query trigger can abort the query.

Post-Query

The `Post-Query` trigger is executed after a query against the database has been executed. A field-level `Post-Query` is executed once for each query, regardless of whether this field will actually be included in the query or not.
Suggested uses for a field-level `Post-Query` trigger:

- Calculate information from the complete result set, like a sum over all records.

Note that this trigger is *not* run when the query is initiated using the set_filter() method of the block programmatically.
The Pre-Query trigger cannot abort the query.

Pre-Change

The `Pre-Change` trigger is executed each time before this field is modified. At the time the Pre-Change trigger is called, the field will still contain the old value.
The Pre-Change trigger cannot abort the change.

Post-Change

The Post-Change trigger is executed each time after the field is modified. At the time the Post-Change trigger is called, the field already contains the new value.
Post-Change triggers cannot abort the change.

Pre-Insert

The `Pre-Insert` trigger is executed before a commit occurs on a record that is pending an insertion. A field-level `Pre-Insert` is executed once for each inserted record.
Suggested uses for a field-level `Pre-Insert` trigger:

- Stamping records with a creation date or created-by value
- Setting a primary key's value
- Setting other hidden, but pertinent, fields with default or pre-calculated values
- Storing historical information in transaction tables

Pre-Update

The `Pre-Update` trigger is executed before a commit occurs on a record that is pending an update. A new or deleted record is not considered "updated" for the purpose of this trigger. A field-level `Pre-Update` is executed once for each changed record, regardless of whether this field actually was changed or not.
Suggested uses for a field-level `Pre-Update` trigger:

- Stamping records with a modification date or modified-by value
- Setting hidden, but pertinent, fields with default or pre-calculated values
- Storing historical information in transaction tables

Pre-Delete

The `Pre-Delete` trigger is executed before a commit occurs on a record that is pending a deletion. A field-level `Pre-Delete` is executed once for each record that has pending deletion.
Suggested uses for a field-level `Pre-Delete` trigger:

- Storing historical information in transaction tables

Pre-Commit

The `Pre-Commit` trigger is executed before a commit occurs on a record. A field-level `Pre-Commit` is executed once for each record that has pending changes, including new and deleted records. A block-level `Pre-Commit` fires prior to a Field's `Pre-Commit` trigger, but after the field-level `Pre-Insert`, `Pre-Update`, and `Pre-Delete` triggers.
Suggested uses for a block-level `Pre-Commit` trigger:

- Stamping modified records with a date or modified-by value
- Setting hidden, but pertinent, fields with default or pre-calculated values
- Storing historical information in transaction tables

Pre-FocusIn

The `Pre-FocusIn` trigger is executed as a new record or field is focused. It is recommended that unless you have a specified understanding of the intention of forms, use `Post-FocusIn` instead of `Pre-FocusIn` as the latter trigger's behavior may change at some point to better reflect record focus.

Post-FocusIn

The `Post-FocusIn` trigger is executed as a new record or field is focused. This may be triggered by a user navigating to a different record, by creating a new record, or by the user changing the focus in the form from one field to another. The actual focus change has occurred when this trigger is fired.
Suggested uses for a field-level `Post-FocusIn` trigger:

- Initialization of fields in some cases where the initialization value depends on what was entered before moving to this field.

Pre-FocusOut

The `Pre-FocusOut` trigger is executed on attempt to move the focus out of a field. This may be triggered by a user navigating to a different record, by creating a new record, by attempting to save the changes, by closing the form, by changing to filter mode, or by moving the UI focus out of the block. The actual record change has not occurred when this trigger is fired. It does not matter whether the operation is performed by the user or programmatically.
Suggested uses for a block-level `Pre-FocusOut` trigger:

- All kinds of field level input validation. It is guaranteed that there is no chance to save data or move the focus out of the field unless the Pre-FocusOut trigger has been successfully run.

Post-FocusOut

The `Post-FocusOut` trigger is executed as a new record is focused in a block. It is recommended that unless you have a specified understanding of the internals of forms, use `Pre-FocusOut` instead of `Post-FocusOut` as the latter trigger's behavior may change at some point to better reflect record focus.

Chapter 5: Working with Fields

Typecasting Fields

Forms supports the following datatypes:

- Text
- Number
- Date
- Time
- Datetime
- Boolean

Default Values

Forms support four different methods for supplying default values to a field.

Default (New Records)

This field property specifies the default value for this field for any new records created. If the field is otherwise editable, then the user can change this value before committing.
In the form XML definition, this attribute is named `default`. It appears on the Designer *Property Editor* as `Default (New Records)`.

Default (Queries)

This field property is similar to Default (New Records), except it provides a default value when in Query mode. If this value is set, and the queryable attribute of a field is False , then you can, in essence, force a condition on all queries. For example, one way to always query records where completed is 'Y', you could have a field named “completed” that is not queryable and has this property set to “Y”. If the field is also queryable, then the value of this property only provides the initial query value – the user could change it prior to executing the query.
In the form XML definition, this attribute is named `queryDefault`.It appears on the Designer *Property Editor* as `Default (Querying)`.

Default to Last Entry

This field property, when set to true, will cause a the field to default to the last entered value for a previous record. This is useful for when a user is doing batch entry, and you want, for example, a date field to continue to default to the last value entered, because the user normally enters the data seqeuntually by date received.
This field property, if true, supercedes Default (New Records) above if the user has previously entered a value for this field.
In the form XML file, this attribute is named `defaultToLast` and accepts a boolean value.It appears on the Designer *Property Editor* as `Default to last entry`.

Setting via Triggers

The last method does not use any field properties, but instead uses a trigger to set default values. Any trigger could be used, but the most common would be either `On-New-Record`, if the default should happen before the user has a chance to enter data; or `Pre-Commit` , `Pre-Update`, or `Pre-Insert` , to set the field to a value after the user has had a chance, but before saving to the database. See the *Trigger Recipes* section for plenty of examples on how to set a default value with dates, timestamps, sequences, etc.

Controlling User Input with Properties

Form fields support several properties that inhibit how the user can enter or change data.

Allow Editing

This field property allows you to specify when, if at all, a field can be modified. The following values are valid:

- **Yes** – Always allow editing.
- **No** – Never allow editing.
- **New Records Only** – Only allow editing of values when the current record is being newly created. This is useful for primary key-like fields, or for audit trails (such as user names, dates, etc.)
- **Update Only –** Only allow editing of values from existing records. Any new records cannot be updated.
- **Null Only** – Only allow editing of values when the field is null. This is similar to *New Records Only*, but also extends editing to existing records, but only if the field was left as empty in prior commits.

Note that any form-level, block-level, and entry-level settings may override the field-level settings.
In the form XML file, this attribute is named `editable` and accepts one of `Y`, `N`, `null`, `update`, and `new`. It appears on the Designer *Property Editor* as `Allow Editing?`.

Required

This property specifies that a record cannot be committed unless this field has a non-empty value.
In the form XML file, this attribute is named `defaultToLast` and accepts a boolean value.It appears on the Designer *Property Editor* as `Default to last entry?`.

Limiting Length of Text

Text fields can have a minimum and a maximum length for entered text. For example, if mimimum is set to 2 and maximum is set to 4, then 'X" would not be a valid entry, but 'XYZ' would be.
In the form XML file, these attributes are named `minLength` and `length`. They appear on the Designer *Property Editor* as `Min Text Length` and `Max Text Length`.
There is no default value for either of these properties.

Trimming Spaces

Fields support left- and right-trimming of entered text. This removes extra trailing spaces from the end of text (*right-trimming*), or leading text at the beginning of text (*left-trimming*).
In the form XML file, these attributes are named `rtrim` and `ltrim`. They appear on the Designer *Property Editor* as `Trim Right Spaces` and `Trim Left Spaces`.
If not explicitly set, then by default, all trailing spaces from user input is trimmed, but leading spaces stay intact.

Controlling User Queries with Properties

Form fields support several properties that inhibit how the user can query data.

Ignore Case on Query

In the form XML file, this attribute is named `ignoreCaseOnQuery` and accepts a boolean value. It appears on the Designer *Property Editor* as `Ignore Case on Query`.

Allow Queries

In the form XML file, this attribute is named `queryable` and accepts a boolean value. It appears on the Designer *Property Editor* as `Allow Queries`.

Chapter 6: Designing for Multiple Interfaces

A form definition, when designed within reasonable guidelines, can be run on a plethora of system architectures and a wide variety of user interfaces. By using the approach taken in this guide, most of your forms will, by default, run on a graphical workstation (X11, Windows, Mac) or in a text-based session (telnet or ssh). This section highlights a few key compatibility issues.

This list, while not exhaustive, should give you a good idea of common portability pitfalls. As with all things in GNUe, you will always have a choice on how to implement your application. GNUe is not about forcing rules on developers, but about providing viable options. There will be instances where the following suggestions simply are not feasible or practical. In any event, these are simply suggestions on getting the most out of Forms.

- **Images**
  Do not make your application dependent on displayed images. It would be acceptable, and appropriate, to display pictures for informational purposes. For example, when doing parts lookups, it would be appropriate to display a picture of the part for reference use. However, it would be normally be inappropriate to prevent the form from working if this image could not be displayed.
- **OS-specific trigger code**
  Python, the default trigger language, provides an extensive library of cross-platform functions. For example, it provides a library of file-access routines that work on all its supported platforms. This is really a broad category as trigger code has all the power of Python behind it.
- **Custom widgets**
  It is often tempting to use a new whiz-bang widget available on a certain platform/widget set. This will surely make your application hard to migrate to other platforms/interfaces, as well as restrict your ability to upgrade to a newer Forms version. Form's widget-set was carefully selected to be as multi-platform-friendly as possible, while still providing all the functionality most forms will need. If your application widgets are not supported by forms, there's a good chance that your form could be more functional with a slight rethinking of its design. Remember: a goal of Forms is to be usable on as many platforms as possible, not to exploit all the features of a particular platform.

Chapter 7: Trigger Recipes

Timestamping a Record prior to a Commit

To automatically fill an entry with a timestamp retrieved from the database, you can use the connection extension getTimestamp(). create a `Pre-Commit` trigger on that block. For example,

```
##
## Pre-Commit [MyBlock]
##

self.MyTimeField.value = MyConn.getTimeStamp()
```

This example assumes your entry is named `MyTimeField` and your connection is named `MyConn`.
As noted elsewhere in this guide, `Pre-Commit` is run prior to saving changes to the database regardless of whether the record in question is being inserted, updated, or deleted. If you want to timestamp only new records, you can use the same code listed above, only inside a `Pre-Insert` trigger. Similarly, if you only want to timestamp modifications, you can use a `Pre-Update` trigger.
By using a timestamp retrieved from the database server, you do not have to worry about differences in the client machines' times. If you would prefer to have the client's time, you can use python's `time` module.

Note: getTimeStamp() is deprecated, it is better to use a server side trigger or use Python's native means of creating a time stamp.

Stamping a Record with User's Login prior to a Commit

To automatically fill an entry with a the user's login name, you can use form.getAuthenticatedUser() and creating a `Pre-Insert` trigger on that block. For example,

```
##
## Pre-Insert [MyBlock]
##

self.CreatedBy.value = form.getAuthenticatedUser()
```

This example assumes your entry is named `CreatedBy`. As noted elsewhere in this guide, `Pre-Insert` is run prior to saving a new record to the database.
This method is commonly called alongside the timestamping recipe above. Together, a Pre-Insert trigger to stamp a new record might look like:

```
##
## Pre-Insert [MyBlock]
##

self.CreatedBy.value = form.getAuthenticatedUser()
self.CreatedOn.value = MyConn.getTimeStamp()
```

See the recipe for timestamping for more information on usage of `getTimeStamp()`.
If your form uses multiple connections, then `form.getAuthenticatedUser()` returns the first login

used. If you want to use a specific connection's login, use the login property of connections:

```
##
## Pre-Insert [MyBlock]
##

self.CreatedBy.value = MyConn.login
```

Auto-Populating an Entry from a Sequence

To automatically fill an entry with a value from a sequence, you can create a Pre-Insert trigger on that entry. For example,

```
##
## Pre-Insert [MyEntry]
##

self.value = MyConn.getSequence("MySequence")
```

This example assumes your entry is named `MyEntry`, your connection is called `MyConn`, and the sequence name as stored in the database is `MySequence`. Note that `MyEntry` is a name originating in your form, whereas `MySequence` is a name originating in your database.

Chapter 8: Invoking gnue-forms

Command line

Depending upon your platform GNUe Forms can either be ran as a command line application called gnue-forms or it may be an icon or menu option someone has setup for your use. We'll cover the basic command line execution of gnue-forms here.

gnue-forms requires that it be passed a GNUe Forms Definition file (a gfd file) at startup. The file could be local or available over the web. Examples

- A gfd file stored on a web server and accessed via a command similar to:

gnue-forms http://www.example.org/sales/customer.gfd

- A gfd file stored on the local hard drive:

gnue-forms C:\forms\cd_collection.gfd

Symlinks

Unix users have a 3rd method of starting gnue-forms via symbolic links (symlinks) to simplify execution of forms. The easiest way to describe this is by example.

Suppose you have a form, contacts.gfd, in the default location (specified in gnue.conf). By placing a symlink called “contacts” somewhere in your path that points back to gnue-forms, you are able to simply run “contacts” and have it run the contacts.gfd form. Or, put more simply, the following:

bash-2.03$ ln -s /usr/local/bin/gnue-forms /usr/local/bin/contacts

bash-2.03$ contacts will display the form /usr/local/gnue/forms/contacts.gfd .

Chapter 9: GNU Free Documentation License

Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human

modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- **A.** Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- **B.** List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- **C.** State on the Title page the name of the publisher of the Modified Version, as the publisher.
- **D.** Preserve all the copyright notices of the Document.
- **E.** Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- **F.** Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified

Version under the terms of this License, in the form shown in the Addendum below.

- **G.** Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- **H.** Include an unaltered copy of this License.
- **I.** Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- **J.** Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- **K.** For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- **L.** Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- **M.** Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- **N.** Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- **O.** Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal

rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (c)  YEAR  YOUR NAME.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled "GNU
Free Documentation License".
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

```
with the Invariant Sections being LIST THEIR TITLES, with the
Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.