Instrument assembly, documentation and release

Fred Wensing, Australian Bureau of Statistics

1. Introduction

Survey instruments usually consist of a package of compiled files that need to be kept together so that the systems can make use of them. While the collection of files that are needed to form this package is not unduly complex, difficulties can arise if one or other elements of such a package are missing or has been incorrectly prepared. For this reason a system has been developed to manage the assembly, packaging and release of population survey instruments at the Australian Bureau of Statistics (ABS).

2. A standard environment

Successful management of instrument preparation is firstly dependent on a standard environment where staff can develop and maintain their programs. The standard environment for instrument preparation at the ABS consists of the following:

- a single network location for all instrument development;
- an agreed folder structure to store all source programs;
- a protected location for all shared source code elements;
- a set of standards and guidelines for instrument design.

2.1 Single network location

A single network location means that instrument code can readily be located by anyone who needs to have access. A single location also means that it is only necessary to have one master copy of shared elements. A single location also means that support and assistance for instrument developers can be more readily provided.

Access to the single network location at the ABS is granted as a part of the process associated with providing access to the Blaise software itself. Therefore, any staff who receives access to the developer installation of Blaise will also have read/write access to the development location.

2.2 Agreed folder structure

An agreed folder structure in the development environment is essential for proper management of source code. The folder structure enables people to readily identify their survey and keep their source code separate from others. Access to survey folders can be restricted to specific staff if required.

At the ABS, a simple structure has been developed whereby each instrument is developed within a folder which exists at the root of the shared network drive. The following simple naming convention is employed to assist with proper identification of instrument folders:

<nickname><reftime><cycle>

where:

<nickname> = a small number of letters (usually 3) for the survey (eg AHS)
<reftime> = two (or more) numbers representing the year or year and month that the survey is to take place (eg 98, 99, 0104)

<cycle> = two or three letters corresponding to the cycle (eg PT1, DR, FIN)

Generally there are only two subfolders below each survey folder, one for local survey-specific modules to be included in the instrument (called INCLUDE) and one for module testing (called MINIMOD).

2.3 Protected location for all shared files and shared source code

In addition to the survey folders there are a number of folders which contain source code that can be included or referenced by all instruments. These folders are write-protected to avoid inadvertent update but are otherwise accessible to all developers.

These shared folders are:

COMMON - a folder containing further sub-folders of common source code. The sub-folders are all identified by a nickname and a version number which relates to the particular component and version release of that component.

CONFIG - a folder containing shared mode libraries, menu and configuration files

EXTERNAL - a folder to hold compiled and loaded Blaise data files which contain look-up lists that have been developed for sharing across surveys.

SYSFILES - a folder of system utility programs, including those used to assemble and prepare instruments

TEMPLATES - a folder to hold templates of programs that can be used by the instrument assembly system to create survey-specific programs (see section 4).

2.4 Illustration of folder structure

An illustration of folder structure implementation at the ABS is given in Figure 1.

```
L:\CONFIG              - for system configuration files
L:\EXTERNAL            - for common look-up data files (code lists etc)
L:\COMMON\HFv3.08      - common modules from v3.08 of the Household form
L:\COMMON\HFv3.09      - common modules from v3.09 of the Household form
L:\COMMON\SYS1.10      - common modules from v1.10 of the system
L:\EXTERNAL            - all coder files named by version
L:\ISS02PT\            - main ISS pilot test project file, instrument and Manipula
L:\ISS02PT\INCLUDE     - ISS pilot test INCLUDE files
L:\ISS02PT\MINIMOD     - ISS pilot test module testing files
L:\ISS02DR\            - main ISS dress rehearsal project file, instrument and Manipula
L:\ISS02DR\INCLUDE     - ISS dress rehearsal INCLUDE files
L:\SYSFILES            - programs used for management of the source code
L:\TEMPLATES           - template or shell programs used to generate standard modules
```

Figure 1. Examples of folders found on the ABS shared network drive

2.5 Standards and guidelines for instrument design

A set of standards and guidelines has been developed at the ABS to assist people with preparation of instruments. These cover various elements such as naming conventions and good practice in Blaise programming, too detailed to be included here. However, the following aspects of ABS standards and guidelines are relevant to instrument assembly:

- all instruments will be prepared in the shared environment;
- all instruments will be prepared using a Blaise Project Definition file which includes the use of version numbering (see section 3);
- all instruments will make use of the standard system modules which include standard type definitions and a common (standard) mode library;
- all instruments will make use of the common (standard) modules for the collection of household composition and demographics (if relevant);
- all modules will include a history section at the top which can be used to track any changes that are made (see section 3);
- all INCLUDE statements will use the relative reference format (rather than the explicit reference format).

2.6 Format for the use of INCLUDE statements

The use of a single network location for instrument source code makes it simple to include instrument modules from anywhere on the network drive. This is achieved through the Blaise INCLUDE statement and a relative reference format.

To include modules of code which are local to the survey, the include statement takes the form:

```
INCLUDE "INCLUDE\<modname>"
```

Where:

<modname> = the name of the module of code to be included

To include modules of code which come from the common folders, the include statement takes the form:

```
INCLUDE "..\COMMON\<release>\<modname>"
```

Where:

<release> = the release version of the common module(s)
<modname> = the name of the module of code to be included

The main reason for using a relative reference format is to avoid the problem which might arise if the drive letter is changed. The relative reference format is also used in preference to the Source Search Path Option in the Project Definition file because the same named module may appear under various release folders.

An example of some include statements found in a typical ABS instrument may be seen in Figure 2.

```
INCLUDE "..\COMMON\PROCS\REPLACEDQ.BLA"
INCLUDE "..\COMMON\SYSv5.04\STDTYPES.BLA"
INCLUDE "..\COMMON\SYSv5.04\SPECS.BLA"
INCLUDE "..\COMMON\HFv6.04\CAIDEMOG.BLA"
INCLUDE "..\COMMON\LFv2.11\INTERVIEWSTATUS.BLA"
INCLUDE "..\COMMON\LFv2.11\STATUSFLDS.BLA"
INCLUDE "INCLUDE\PersonQre.bla"
```

Figure 2. Typical include statements found in ABS instruments

3. Version control at both module and instrument levels

In order to manage the succession of changes that can happen to an instrument, a system of version control has been implemented at the ABS.

Not only are instruments at the ABS given a version number but each module of source code is required to have provision for the recording of version history which is to be updated whenever that module changes.

3.1 Instrument version numbering through the Blaise Project Definition file

The Blaise Project Definition file enables a composite version number to be added to an instrument through the Project Options menu/Version Info tab. This composite version number is a combination of Major and Minor version plus Release and Build number (see figure 3).

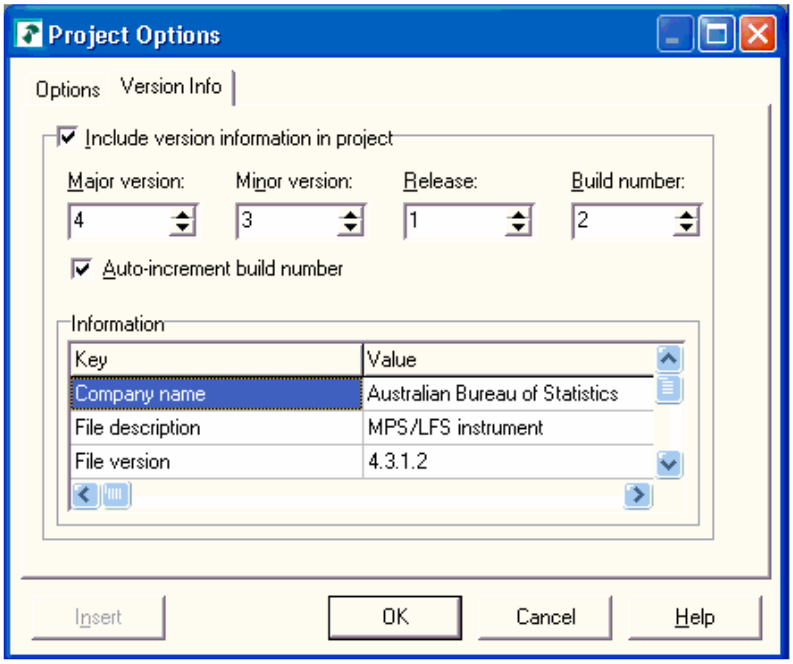


Figure 3. Version information found under Project Options

Local rules are devised for each survey to enable people to decide which number to change. It is also good practice to ensure that the Auto-increment build number option is turned on so that each compilation of an instrument (for whatever reason) will have a new build number and therefore have a unique version number.

The main version rule, which is followed for all surveys, is that a new release must always receive a new number. In most cases the new number will involve a change to the Major, Minor or Release number, unless it is definitely known that the new compilation will produce an instrument which is

fully compatible with the previous release. In such cases, a change will occur in the build number only.

3.2 Accessing the version number from the compiled instrument or data file

One advantage of incorporating a version number into an instrument, is that the version number of both an instrument and the collected data can be subsequently identified. This is done through the Help/Info menu which then brings up the Dialog shown in Figure 4.

Figure 4. Help info dialog

To access the version information for either the Metadata or the Data press the relevant button of the info dialog. An information screen such as that shown in Figure 5 will then be displayed.

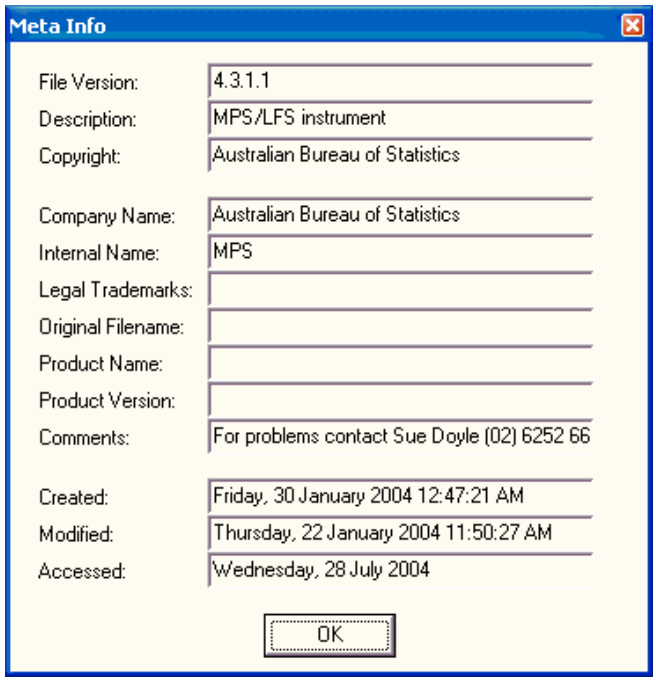


Figure 5. Version information for the metadata

3.3 Maintaining a history of module changes

Because instruments are usually composed of modules of source code, it is important that a history of the changes to those modules is maintained.

For ABS instruments a simple system of history recording has been implemented which enables all changes to be identified and dated. The system involves the use of a comment placed at the top of each unique module of code. The format of that history is such that information from it can be extracted by the instrument preparation utility.

The general format of the history is that of a Blaise comment in which there are some lines of free-form text to identify the module, its main author and its main purpose. Following that information, provision is made for update history in the format:

<Version> <dd/mm/yyyy> <Person> <Change>

where:

<Version> = a decimal number that identifies a major and minor version number for any change
<dd/mm/yyyy> = a properly formatted date
<Person> = the unique user name of the person making the change
<Change> = a brief description of the change

An example of a properly formatted history is shown in Figure 6.

```
{*********************************************************
 Module Name            :  MPHS04FIN.BLA
 Author                 :  Fred Wensing
 Update History:
 Vsn  dd/mm/yyyy Person Change
 ---- ---------- ------ ------
 2.05 26/05/2004 Halls  - Add derivation for HoursMainJob
                        - Deleted edits WTW1, WNM1 and WNM3
 2.13 21/06/2004 doylsu - changed 'includes' to point to COMMON
 2.14 23/06/2004 doylsu - released with v1.063 using August MPS
*********************************************************}
```

Figure 6. Example of formatted history

4. Templates used for standard programs

Survey systems, which are required to support a variety of surveys, ultimately end up using standardised instruments, programs and practices. Despite the fact that standardised programs are used, however, those programs often need to be modified (eg. to point to new survey instruments) and recompiled before use. While the nature of the day-to-day changes to the standard programs is often small (eg: a changed instrument name; a changed label) there is a risk that facilities will not work propery, or fail altogether, if the changes are not made correctly.

For this reason, a system of template programs has been developed at the ABS in which certain sections of those programs are updated by the system before use. The template programs are, for all intents and purposes, functional programs in which particular sections of the source code have been tagged for modification or replacement by the assembly system.

The general method of tagging is to insert some identifiable and formatted text within a comment in the source code. This text can be located by the assembly system which scans the source program(s) for the tags and takes appropriate action.

Two forms of tagging have been devised:

- tagging of a line to be replaced;
- tagging of a segment of code to be included, removed or replaced.

To complete the assembly system, details of tag replacement information are stored in a tag definition dataset. Reference is made to this dataset during the assembly process.

4.1 Tagging of a line to be replaced

The general format of a tag for a line to be replaced is:

{>>> <tagname>}

Where:

{}	= the standard delimiters for a comment in Blaise or Manipula
<tagname>	= a text string of 5 characters which identifies the tag by name
>>>	= an identifying characteristic.

The above tag is placed at the start of any line which is to be replaced. When the assembly system finds such a line it rewrites the remainder of the line (after the tag) using information found in the tag definition dataset.

All comment lines which contain the >>> characteristic are assumed by the assembly system to be a tag of some kind. This set of characters was selected because it can never appear as part of a Blaise or Manipula program. While this set of characters could appear within a comment, for other reasons, the assembly system also needs to recognise the tag name before taking any action to modify the line of source code. The set of characters is also readily recognised by persons looking at the program source code.

Note that the same tag name can be used multiple times within an instrument if more than one line is to be changed in the same way.

Figure 7A shows a section of Manipula program which contains a line that has a tag named META1 to identify a metadata reference (the template currently has **'LFS211'** on that line).

```
...
USES
   SurveyMeta
{>>> META1} 'LFS211'
   DATAMODEL mList
      Fields
         xPSU      : STRING[5]
...
```

Figure 7A. Manipula program showing a line tag before update

Figure 7B shows the same section of Manipula program, containing a new metadata reference for META1 (now **'LFS403'**) after it has been processed by the assembly system.

```
...
USES
   SurveyMeta
{>>> META1} 'LFS403'
   DATAMODEL mList
      Fields
         xPSU      : STRING[5]
...
```

Figure 7B. Manipula program showing a line tag after update

4.2 Tagging of a segment of code to be included, removed or replaced

When a segment of code is to be included in a program then the tag is written in the form:

{>>> INCLUDE <inclname>}

Where:

{} = the standard delimiters for a comment in Blaise or Manipula
INCLUDE = a keyword used by the assembly system
<inclname> = a text string to identify the segment of code to be included
>>> = an identifying characteristic

When the assembly system encounters such a tag it will remove the tagged line and insert the lines of text (usually progam text) from a file which has the name **<inclname>.INC**. The include file is expected to be in the same folder as the program being updated.

When a segment of code is 'included' in this way the system adds two new tags to the program so that the segment of code can be removed by the assembly system at a later date. The two new tags are:

{>>> START <inclname>}

and

{>>> ENDOF <inclname>}

Where:

{} = the standard delimiters for a comment in Blaise or Manipula
START = a keyword used by the assembly system
ENDOF = a keyword used by the assembly system
<inclname> = a text string to identify the segment of code which has been included
>>> = an identifying characteristic.

These tags are added to the start and end of any segment of code which has been included.

Figure 8A shows a section of a Blaise instrument which contains a tag to include a segment of code named **Suppfields.INC**. Note that the use of <<< at the end of the tag definition serves no functional purpose but is done for appearances.

```
    xNum2 : TNum99
    xChar1 : TString1
    xChar2 : TString2
{>>> INCLUDE SuppFields <<<}
    AUXFIELDS
        aName, aNamePoss, aIsAreLC, aIsAreUC, aHasHaveLC,
        aHasHaveUC, aWasWereLC, aWasWereUC,
        aDoesDoLC, aDoesDoUC, aVerbEnd
            : STRING
```

Figure 8A. Code segment showing the INCLUDE tag

Figure 8B shows the same section of a Blaise instrument with the included **Suppfields.INC** code and the additional tags which mark the start and end of the included code.

```
    xNum2 : TNum99
    xChar1 : TString1
    xChar2 : TString2
{>>> START SuppFields <<<} {v3.14 04/04/2004}
    INCLUDE "INCLUDE\Introduction.bla"
    FIELDS
        Introduction: BIntroduction
    INCLUDE "INCLUDE\Usualwork.bla"
    FIELDS
        Usualwork : BUsualwork
    INCLUDE "INCLUDE\Absences.bla"
    FIELDS
        Absences : BAbsences
    INCLUDE "INCLUDE\Childcare.bla"
    FIELDS
        Childcare : BChildcare
{>>> ENDOF SuppFields <<<}
    AUXFIELDS
        aName, aNamePoss, aIsAreLC, aIsAreUC, aHasHaveLC,
        aHasHaveUC, aWasWereLC, aWasWereUC,
        aDoesDoLC, aDoesDoUC, aVerbEnd
            : STRING
```

Figure 8B. Code segment showing the text included and the tags for START and ENDOF

Note that the assembly system has added version information, transferred from the history header of the main program, as a comment to the included code. This is so that staff looking at the code subsequently are made aware of when the code was added. Once again, the use of <<< at the end of the tag definition has been done for appearances only.

4.3 Further discussion of the tag approach

The tag approach was devised in answer to a need to be able to include or replace small or large segments of code in a program. Blaise provides an INCLUDE statement for use within instrument definitions but it is limited to complete sets of fields and/or rules only. In addition there is no provision for an INCLUDE statement in the Manipula language.

The advent of prepare directives (from version 4.6 of Blaise onwards) may provide an alternative to the tag approach described in this paper. However, at the time that this system was developed, prepare directives were not available in Blaise.

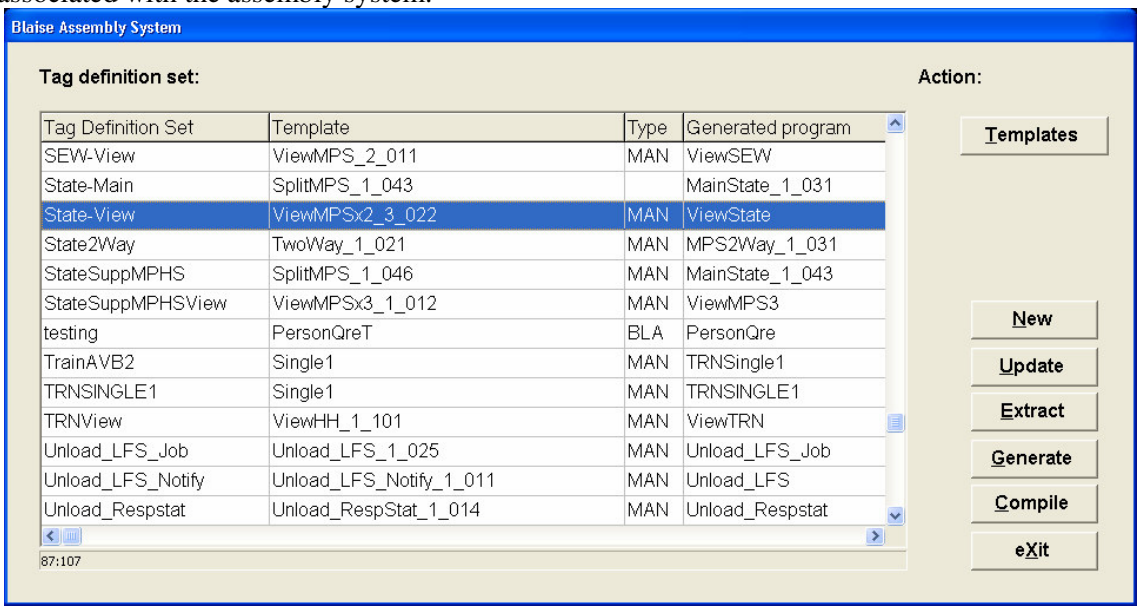
5. Assembly utility

With the existence of template programs, that are suitably tagged, it is possible to generate usable copies of the templates in which the tagged elements have been replaced or updated accordingly. To do this an assembly system has been built, in Maniplus and Manipula, containing the following elements:

- a register of templates
- a register containing tag set definitions
- an interface to enable the registration of templates and updating of tag set definitions
- a set of procedures associated with the generation of programs from templates.

5.1 Interface to the Assembly system

A simple Maniplus interface (see Figure 9) is used to display a list of tag set definitions for the assembly system. The interface also contains a series of buttons to activate the various procedures associated with the assembly system.



Tag Definition Set	Template	Type	Generated program
SEW-View	ViewMPS_2_011	MAN	ViewSEW
State-Main	SplitMPS_1_043		MainState_1_031
State-View	ViewMPSx2_3_022	MAN	ViewState
State2Way	TwoWay_1_021	MAN	MPS2Way_1_031
StateSuppMPHS	SplitMPS_1_046	MAN	MainState_1_043
StateSuppMPHSView	ViewMPSx3_1_012	MAN	ViewMPS3
testing	PersonQreT	BLA	PersonQre
TrainAVB2	Single1	MAN	TRNSingle1
TRNSINGLE1	Single1	MAN	TRNSINGLE1
TRNView	ViewHH_1_101	MAN	ViewTRN
Unload_LFS_Job	Unload_LFS_1_025	MAN	Unload_LFS_Job
Unload_LFS_Notify	Unload_LFS_Notify_1_011	MAN	Unload_LFS
Unload_Respstat	Unload_RespStat_1_014	MAN	Unload_Respstat

Figure 9. Assembly system interface

5.2 Register of templates

Template programs are assigned a unique identity consisting of a name and version number (Major, Minor and Release number combined), and registered in a simple Blaise database. Other information included in the register is the type of template (Blaise or Manipula) and a brief description.

Once they are registered in that database they can be linked to any tag set definition.

5.3 Register of tag set definitions

The register of tag set definitions contains the information required by the assembly system to modify one or more of the tags it finds within a given template.

Figure 10 shows an example of a tag set definition in which the values are given for the various tags which are expected within the template.

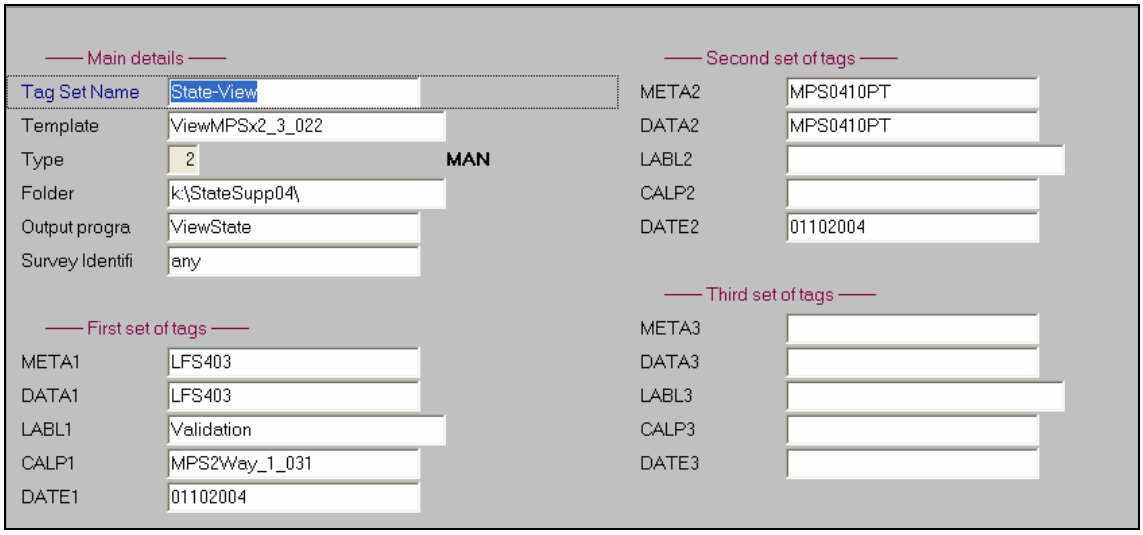


Figure 10. Example of a tag set definition

In this example, the **State-View** tag set definition contains the tag values to be used with the template called **ViewMPSx2_3_022** to produce an output program called **ViewState.MAN**. You can see that the tag called META1 will be set to the value **`'LFS403'`** when processed by the Assembly system. This is the origin of the tag value used in Figure 7B.

Note that tags which are not used in the corresponding template are left blank in the tag set definition.

5.4 Set of procedures associated with the generation of programs

There are two main procedures which have been developed to support the assembly system:

- procedure to generate an operational program from a Template (using the selected tag set definition)
- procedure to extract the included segments of code from an existing operational program.

These procedures are maintained as Manipula programs that are called by pressing the appropriate button in the interface.

The other procedures in the assembly system enable the tag sets and the template definitions to be updated.

5.5 Operation of the Assembly system

The usual mode of operation of this system is for staff to activate the interface and generate program(s) as and when required.

Special provision has been made, however, for the generation process also to be run by the survey system itself, via a scripted call to the generation Manipula program. This is achieved by setting up the tag set definition in a special way that enables the values of tags to be passed to the generation program through parameters.

The advantage of this assembly system is that tailored programs to support the operations of ABS surveys can be quickly and reliably produced.

6. Instrument package utility

Once all the instruments and associated programs (some generated) exist for a survey, it is necessary to collate the compiled files into a package for installation into the survey systems. At the same time, it is important to collate the contributing source files into another package for archival purposes and to assist with any problems that may arise.

An instrument preparation utility (called PrepareZips), built mainly in Maniplus and Manipula, has been produced at the ABS to take care of the packaging process. The main features of the utility are:

- a register of instrument release details;
- an interface to manage the steps involved;
- enforcement of version control;
- preparation of zip files containing various parts of the instrument;
- a list of contributing source code drawn from the common (shared) store;
- a report of the version history of all local contributing modules of source code;
- a listing of the edits found in the source code.

6.1 A register of instrument releases

Each release is recorded in a register of instruments. The registry entry records the type of release (instrument, code list, template etc.), the version number for the release and identifies the files (Blaise datamodels, Manipula and text files) to be included.

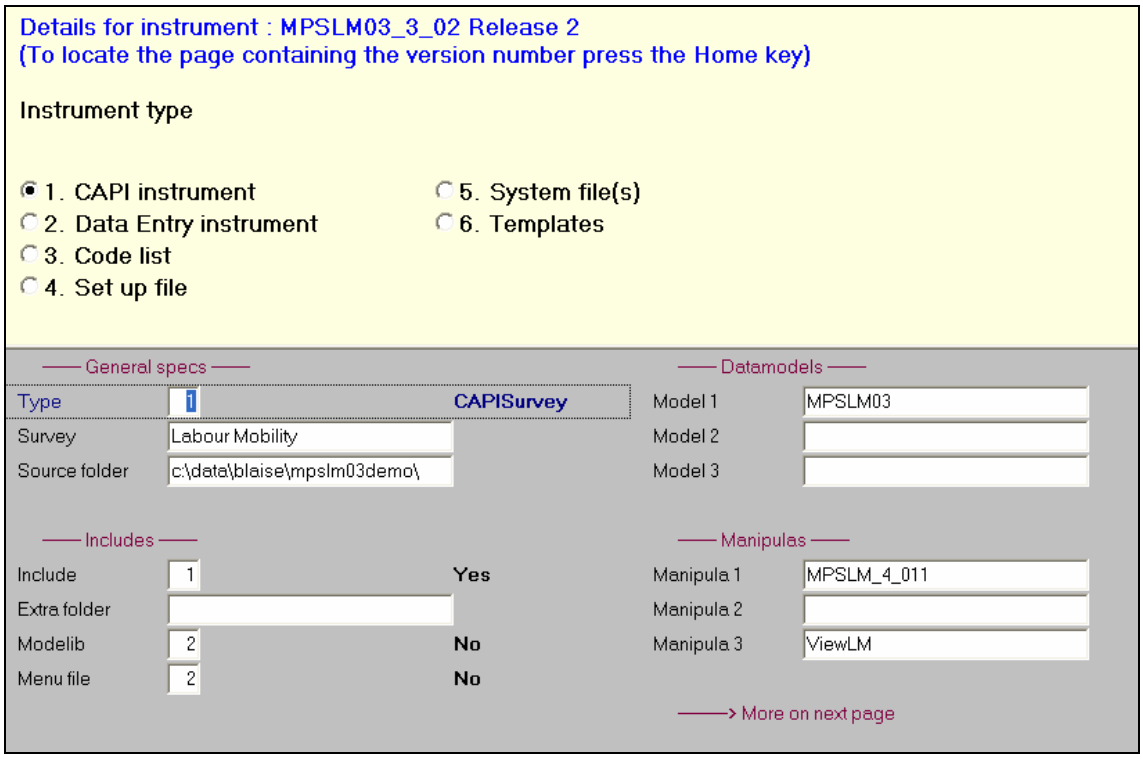


Figure 11. First page of a typical entry in the instrument register

6.2 An interface for the preparation utility

The interface for the instrument preparation utility (see Figure 12) shows a list of the current released instruments and a series of buttons which activate the various steps.

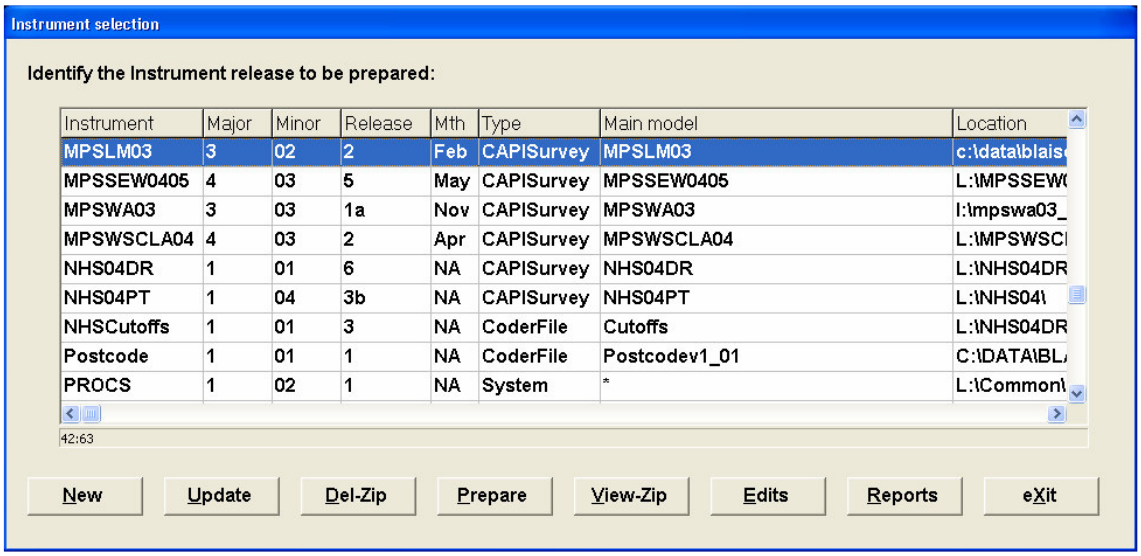


Figure 12. Interface for the preparation utility

6.3 Enforcement of version control

When an instrument is registered for preparation the utility requires details of the major, minor and release version numbers. Without these details, a release cannot be prepared.

Once supplied, the release number is checked against the release number found in the Blaise Project Definition file. If the version number is not the same then an error message is produced (see figure 13) and the preparation stops.

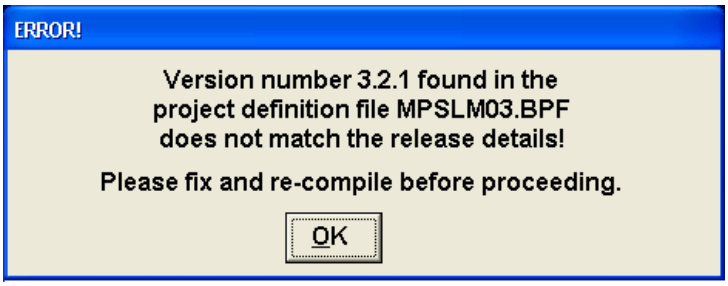


Figure 13. Error message for mismatching version number

6.4 Preparation of the Zip files

The preparation utility produces a list of the full names of all the required files and then activates Winzip to load two zip files:

- one containing the compiled programs along with any data files or text files that are needed;
- a second, containing the local (survey specific) contributing modules of source code for archival purposes.

The utility is programmed to take note of the instrument type (instrument, code list, template etc.) and adjust the lists of files accordingly.

The Zip files produced are given a name which matches the instrument name and full version number. This makes it very easy to align the Zip files with their corresponding entry in the instrument register.

Note that only the local contributing modules of source code are loaded into the source code Zip file because the common (shared) modules would have been released in their own right and would already have been loaded into their own source code Zip file.

6.5 Report on the contributing source code

The instrument preparation utility produces a report (see Figure 14) of the contributing modules of common source code. This list can be used for checking purposes and can assist any person who wishes to re-create the instrument from the source code.

```
ZIP file : MPSLM03_3_022_Feb_Source
-----------------------------------
List of shared components used

TYPE=Include
------------
..\COMMON\HFv7.03\
        Caidemog.BLA
        Caihhold.BLA
..\COMMON\LFv2.14\
        Bhourswork.BLA
        Blastjob.BLA
        Blastjobft.BLA
        Bunempnilf.BLA
..\COMMON\PROCS\
        Firstletteruc.BLA
        Getdatefills.BLA
..\COMMON\SYSv5.10\
        Householdid.BLA
        Iwmsflds.BLA
        Officefields.BLA
        Specs.BLA
        Stdtypes.BLA

TYPE=Modelib
------------
..\Config\
        Mode2_f14_g9x2_desc_tdesc.BML
```

Figure 14. Report of contributing modules

6.6 Report of the version history of contributing modules

The instrument preparation utility also scans all the local contributing modules of source code for the version history and produces a report of the latest changes (see Figure 15). This report, which shows the version number and persons who have made the latest changes, provides a simple way of identifying the instrument changes over time.

This report is only possible if the history entries have been formatted in accordance with the description given in Section 3.3.

```
ZIP file : MPSLM03_3_022_Feb_Source
-----------------------------------
List of modules and history

NAME                          LINES  VERSION     DATE      AUTHOR
==========================  ======  =======  ==========  ======

TYPE=BLA
--------
Education                       168  3.02     04-02-2003  mcnall
Initial_sequence                 83  3.02     07-04-2003  mcnall
Introduction                     84  3.02     04-02-2003  mcnall
Job_mobility_other              117  3.02     07-04-2003  mcnall
Mpslm03                        1502  2.13     01-08-2003  Doylsu
Occ_prev_job                    125  3.03     03-04-2003  MCNALL
Personqre                       950  3.12     06-08-2003  doylsu
Previous_job_status             170  3.02     29-01-2002  mcnall
Specsrules                       48  4.13     06-08-2003  Doylsu
                              ------
                              8900

TYPE=MAN
--------
Viewlm                          609  1.14     21-02-2003  Wensfr
                              ------
                                609
```

Figure 15. Report of instrument version history

6.7 Listing of the edits found in the source code

The instrument preparation utility can also produce a list of the edit statements in the local contributing modules of source code (see Figure 16).

This listing is produced using a SAS program (developed in 1996) which scans the source code for the Blaise keywords SIGNAL and CHECK and then lists the statements which follow. For this aspect of the utility to operate successfully it is important that these keywords accompany each edit.

```
MPSLM03_3_022_Feb_Source                              July 31, 2004

List of Blaise programs containing SIGNAL or CHECK
in C:\DATA\
                                                       Line
NAME                              SIGNAL    CHECK     count
Education.bla                        0         0       168
MPSLM03.bla                          0        20       283
...
SpecsRules.bla                       0         0        48
                                  ======    =====     =====
                                     0       161      8900
*********************************************************
MPSLM03_3_022_Feb_Source
List of edits found
=========================================================
Module : MPSLM03.bla
---> Line 627
     CHECK
     Hhld.Qns.Scope[LCount].xPersTypePD <> Visitor
     "Visitors are not included in this survey - select UR or Remove"
---> Line 942
     CHECK
     Hhld.Qns.D3[LCount].Age <= 24
     "Boarding school pupils must be aged 24 years or less"
=========================================================
Module : PersonQre.bla
---> Line 556
     CHECK
     pYearArrived <= YEAR(xIntDate)
     "Cannot have arrived in Australia after this year"
...
```

Figure 16. Example of a list of edits found in an instrument

7. Conclusion

The systems described in this paper have been developed in order to ensure that instrument packages cans be readily and reliably produced. When an instrument package is produced, the local source code is checked for version and history information. The local source code for an instrument package is also collated into a separate Zip file for storage purposes. Summary documentation is then produced to identify the contents of the instrument and the current version information.

The success of these facilities is dependent on good practice in the management and sharing of code, a simple system for recording update history and the implementation of version control. Added to this is a special method of identifying or tagging lines of code which can then be updated by the system if required.