CODING TRICKS TO SAVE RESOURCES

Barbara S. Bibb, Lillie Barber, Ansu Koshy, Research Triangle Institute (RTI), USA

Introduction

As computer aided survey administration has developed, survey instruments have become increasingly complex. Increased complexity in survey methodology seeks to gather the most correct data possible. For surveys to be appropriately administered under all possible circumstances, instruments are customized for individual respondents, reducing burden and complexity for the respondent but raising it for the programmer. As a result, the software applications used to present the instruments can be pushed to their limits. Innovative methods are therefore needed so that instruments can be built within the application's limitations and so they use resources conservatively.

This paper will discuss a Blaise instrument that would have grown out of control using straightforward instrument coding methods, but which was rescued by some tricks applied during development. The survey was about finances and insurance, difficult subjects at best for most people, and very difficult for respondents whose personal lives took many twists and turns. To elicit good answers, the questionnaire designers included loops, cross-references, recall aids and other techniques to help obtain the necessary data. This may have been helpful for the interviewer and subject, but caused difficulty during implementation.

For example, in the early stages of programming, one particularly complex module was only about a quarter complete and contained over 15,000 lines of code. Because of its length, simple "typo" coding mistakes would have been impossible to find, and there was concern that the program would break with each compile. This paper will present some of the ideas and constructs that were used to reduce the amount of code necessary to complete the instrument. Pros and cons of the techniques used will be discussed as well as some lessons learned.

Historical Background

In the middle of the 1900's census taking and simply counting was extended to create survey research by using questionnaires and statistical surveys to collect data about people and their attitudes, opinions, and beliefs. The effects of radio were investigated by collecting public opinion data to gauge reactions to and changes in the economic environment. At the University of Michigan, Rensis Likert and Angus Campbell (former employees of the U.S. Department of Agriculture's Division of Program Surveys), formed the Survey Research Center (SRC). Survey research expanded to cover behaviors and elements of interest in social sciences.

During this period, data was collected using complex paper forms containing directions that guided respondents around inappropriate questions. Later, computers made it possible for surveys to be customized for individual respondents, with obviously inappropriate questions automatically skipped. More sophisticated computer applications applied complex logic to create surveys that seemed to talk to the respondent directly, personalizing questions based on the respondent's answers and demographic characteristics. Thus, an effectively programmed instrument reduced the burden on the respondent.

As computer storage became more cost effective, more data could be collected while the effective programming of the instrument worked to reduce the burden on the respondent. In addition to the data collected from a respondent, computer applications have been asked to generate and maintain variables

used to appropriately guide the respondent through the maze of questions, administering only those appropriate for that respondent and his/her situation. The number of variables collected and stored within a survey instrument can rapidly increase when the instrument is designed to collect and maintain person level data within a household data collection effort. To be sure that no one in the household is left out the array sizes specified are large. Space must be allocated for each possible array element whether or not that space is needed in a particular administration. Application programs usually allow ample flexibility, but even so innovative projects push the limits.

Project Background

To test methodology in a recent project for the US Census Bureau, an experimental version of some standard questions was programmed. The respondent for the study was the person of appropriate age who answered the phone or was available to come to the phone. This person answered questions for the household and for each household member, so data being collected was both household level data and person level data. The person-level data was stored in arrays where the index of the array was mapped to each person's position on the household roster.

In addition to a fixed set of modules, the program contained three modules that each collected the same data in different ways. Two of these modules were question sets from existing studies. The third was an experimental module that was implemented for the first time. This module is the topic of this paper. Each respondent went through only one of these three modules, providing control groups and data for comparison. The project goal was to try to determine which set of questions would generate the most accurate (reliable and valid) data. The programming challenge was to code the experimental module.

Because human memory can be inaccurate, the experimental module was designed to collect information by repeating some questions a number of times in slightly different contexts. The respondent answered the sequence of questions in a number of ways, eventually covering all household members. Although sections of questions were repeated, the total respondent burden was reduced because each round of questions collected as much data as possible. In the later rounds, the only repeated questions were those needed to pick up data missing as a result of the previous question sequences.

In the worst case, the entire series of questions would be repeated for each household member. The program, however, reduced respondent burden by asking a question and then asking which household members could be included as having the same response. Flags within the program allowed the respondent to answer for all household members with different circumstances and prevented needless repetition of questions. To ensure accuracy of the data with respect to the respondent's memory of the past, questions to verify the data were added.

Programming began with a standard straightforward approach of asking the questions by grouping them into blocks that could be re-used when the question sequences needed to be repeated, setting the flags, and then applying logic to determine which sequences of questions needed to be asked next. To correctly drive the program, the backend variables needed to be set. Between the repetitious questions and the required backend variables, the section of the program that was doing this work rapidly became very large. There was concern that with each attempt to compile, the internal Blaise limits would be exceeded and the project would die. In addition, the path changed as the backend global variables were adjusted and updated with additional question responses.

To include every person in each household, this project had especially large person-level arrays. The project allowed for up to 16 household members. Therefore, each person-level array had an index of 16,

where the index number corresponded to the household member's position on the original roster enumerating the household.

To determine the status of each household member with respect to the data being collected, global "flag" variables were defined for each person. Data was being collected by month and there were 17 applicable months. In addition, there were 14 different classifications of the data or types of insurance coverage. Backend internal arrays were filled with variables needed to summarize the collected data and drive the program. The declarations of these internal person level arrays initially resulted in 3808 variables (16 household members * 14 types * 17 months). This variable count does not include any of the answers to the questions presented to the respondent that were needed to collect the data. The enormous number of variables presented problems. A program with such large numbers of variables could require more space than Blaise, the application program, allows. Blaise has internal limits that include both the size of the code files the program is able to compile and the number of variables that Blaise can maintain and can keep track of. Any application program would have such limitations.

To resolve the problems in handling this experimental module, the programming staff decided that it was essential to simplify the code if possible. This programming exercise had two challenges:

- To reduce the number of variables
- To reduce the number of lines of code

Before re-thinking the initial approach, one file contained over 15,000 lines of code, much of it repeated with only a change of index. That file only covered a quarter of the intended question sequences for the module. There was too much code to process, debug, and work with. Typos within the indices or any other part of a code file this size would never be found. After innovations were applied, the final module was reduced to two files of about 4000 lines.

Block development

Several steps were taken to try to simplify the code. Repeated questions were grouped into blocks which was a first step in reducing the number of lines of code. Each block could be repeated as necessary to collect the appropriate data for each household member. The paths through the blocks depended on which household members the respondent was answering for with the current question set as well as some global counters. Thus the block-level structure became more complex, but the body of code itself became simpler.

Introductory questions were grouped in one block, and parameters were used to pass appropriate data into the blocks of follow-up questions. As data was collected global counters were used to help determine the correct path. Global counters can create problems. For example, each question set was restricted for each respondent. The questions could only be repeated twice, so that no one was asked about having any more than two kinds of insurance. The number of insurances reported for each person was held in a global counter. Once that counter reached its limit, code was to be skipped. These counters were used in this application to determine the path through the blocks and whether or not a block should repeat, but they were dynamic. They changed as questions were answered by the respondent.

The global counters created two opposing problems. As the counter increased, certain sections of code were to be skipped which created a path change. Entry into a block depended on not having been there

before and/or on the global counter as well as on the answers to the previously presented questions. As the counters were incremented, question sets moved from “on path” to “off path”. This change in path without any intervention will lead to the loss of the data initially collected.

The second problem the global counter created is the counter example of the problem discussed above. Since the paths were dependent on the global counters, some instances caused a question to be added to the path from a block that was previously presented and presumed complete. The program would skip back to a question unexpectedly in that “completed” block once a counter increased to the requirements in the logic. To correct the presentation of extraneous questions from completed blocks and to prevent path change of previously presented blocks, if and keep statements were applied to the blocks. The block was only called if the “end” marker was empty. Once the “end” marker was encountered on the path and was answered the block was protected with a keep statement.

```
If end_block = empty then
      Present the block
Else
      Keep the block
Endif
```

This type of flow control is well documented in the Blaise literature discussing the protection of blocks from further change. (Statistics Netherlands (1999) Blaise Developers Guide, Blaise for windows 4.1, A Survey Processing System . p. 195). It is common to protect rosters and tables from change by the interviewer. We found that unexpected change created by the use of global counters and variables could be managed with the same techniques. To prevent the loss of previously collected data, the block calls were placed in ‘if’ statements with keep statements following the ‘else’ condition. In other words, when the questions were asked as part of the “if” code they were on-path, and when they were skipped within the “else” condition, data were protected by the keep statement.

As data was collected, this implementation essentially locked the administered blocks. The locking of these blocks prevented the interviewer from backing up to make corrections or adjustments but prevented loss of data regardless of which path was followed. The program went to production with this access-limitation approach in place.

Number of variables

As program design progressed, the size of the program and the number of backend variables quickly became large. Many of these variables were defined on a global level so that they would be available to all blocks and all modules. It was obvious that the number of variables should be reduced if at all possible so that the internal storage required within the Blaise application would not become so large that it would reach the application’s limits. To this end, sets of classification variables were replaced with a single string variable. One was defined for each set of classification variables (14 of them), where the classification variable indicated a type of insurance coverage. A 15th classification variable was created and called “master”. This master indicated if there was any insurance coverage of any type. These variables were defined as strings of length 17. Each column of the string represented a month and covered more than one year. Doing this reduced the number of global variables being stored for this particular piece of the project from 3808 variables (16 household members * 14 types * 17 months) to a mere 240 variables (16 household members * 15 types), a reduction in complexity within the instrument and more importantly on the resulting analytic dataset.

The string method for each classification variable was a very effective and novel approach to the problem of too many variables. Each column of the string represented a month. A method was devised to enable the columns to be set on and off as appropriate. Changing a single character in the string replaced setting an individual variable on or off. For example, the third character of the string represented the month of March. If the respondent answered "yes" for a month, such as by saying they had insurance coverage that month, the third character would be set to "Y". For this particular project, once a month was "on" it would not be turned off.

The 15th type, the master, was set if any of the types were set for a particular month. This allowed the program to evaluate a single variable to determine the insurance status of the respondent or of anyone in the household. Each household member had their own set of insurance "strings".

Type1(i) := "NNNNNNNNNNNNNNNNN" initialized the entire string of months if type1 was indicated for the i th household member. At the same time the master for the i th household member was initialized.

Master(i) := "NNNNNNNNNNNNNNNNN"

Strings were left empty for all kinds of insurance that the respondent indicated were not used by a particular household member.

A sequence of questions asked to determine if any household members were covered by each particular type of insurance. Initializing the string to all off was the first step in capturing the data about the insurances carried by members of the household. A range of dates was captured through the question sequence, so the problem that remained was to collect the information together and set the appropriate columns of the strings from 'N' for no insurance that month to 'Y' for covered by insurance that month. The position of the column translated to the month covered.

A procedure was written to set the columns (months of coverage) of each string where the string represented the type of insurance (one of the 14 string variables). This procedure was also used to set the master string, representing coverage by any insurance carrier. The procedure used parameters generated from the collected data to turn appropriate columns on. Parameters were required. Inside nested procedures, parameters are the only way to be sure that the procedure is acting on the correct data. The procedure TypeSTR called a second procedure which set an individual column.

```
PROCEDURE TypeSTR
  PARAMETERS
    TRANSIT typestr : string
    TRANSIT MASTER : STRING
    IMPORT m : INTEGER {person index}
    IMPORT BMON : INTEGER
    IMPORT BYR : integer
    IMPORT EMON : INTEGER
    IMPORT EYR : integer
    locals j:integer
           k:integer

     RULES
{the j loop sets the first 12 columns of the insurance string }
     for j := 1 to 12 do
     set_ FLAG(Typestr, J, j, 2009, bmon ,byr, emon , eyr )
     set_ FLAG(MASTER, J, j, 2009, bmon ,byr, emon , eyr )
     enddo
```

```
{the k loop sets the last 5 columns of the string}
     for k := 1 to 5  do
     set_ FLAG(typestr, k+12, k, 2010, bmon ,byr, emon , eyr )
     set_ FLAG(MASTER, k+12, k, 2010, bmon ,byr, emon , eyr )
     enddo
ENDPROCEDURE
```

The first procedure, TypeSTR, used indices and affected the entire string. It called a second procedure, SET_ FLAG, which set only one column of the string to 'Y'. The loop in the first procedure called the second procedure to set the single column the number of times indicated by the data collected. Since the string variable for this project covered more than one year (17 months), the indices were defined to cover the first year (12 months) and then the next 5 months.

```
PROCEDURE SET_ FLAG
  PARAMETERS
{The variable string being set} TRANSIT FLAG : string
    loc :integer
    flagmon : integer
    flagyr : integer
{beginning month and year being set}
    IMPORT BMON : integer
    IMPORT BYR : integer
{ending month and year being set}
    IMPORT EMON : integer
    IMPORT EYR : integer

  AUXFIELDS
    BTmpDate : DATETYPE, EMPTY
    ETmpDate : DATETYPE, EMPTY
    refdate :  DATETYPE, EMPTY
    m : integer
    n : integer

  RULES
{make dates of parameters being passed}
    BTmpDate := TOdate(BYR,BMON,1)
    ETmpDate := TOdate(EYR,EMON,1)
{define a reference date that is today}
    refdate := todate (flagyr, flagmon,1)
    m := loc-1
    n := loc+1
{if the months being passed are within the dates to be set}
if refdate >= BtmpDate and refdate <= Etmpdate then
    if  loc = 1 then
{set first column}
          FLAG := 'Y' + substring(FLAG,2,17)
       elseif loc >1 and loc < 17 then
{set middle columns}
          FLAG := substring(FLAG,1,m) + 'Y' + substring(FLAG, n,17)
       elseif loc = 17 then
{set last column}
          FLAG := substring(FLAG,1,16) + 'Y'
       endif
endif
ENDPROCEDURE {SET_ FLAG}
```

The SET_ FLAG procedure used the parameters sent by the TypeSTR procedure. It converted the parameters to dates to compare to a reference date. If the dates were within range, it used the parameter j to turn the j^{th} column on. SET_ FLAG is called from a loop and will be called 17 times to change the column values appropriately, one at a time.

Conclusions

Survey application software can be pushed to its internal limits as the complexity of survey instruments increases, requiring creative thinking to meet the demands of the methodologists and use the applications on hand. Some of the tricks developed on this project allowed an increase in instrument complexity without negatively impacting the survey application.

The programming changes we discussed allowed the project to be successfully completed in spite of high complexity. Several lessons stood out from our experience:

- Programmers should carefully review the use of blocks of code and global variables.
- Our use of blocks of code was not unusual, but the repeated use of the same blocks caused some unforeseen issues that needed to be addressed. These issues were solved by using procedures and parameters.
- By redefining sets of variables to a single string variable, the number of variables being handled became manageable. Complexity of maintaining these individual string variables was handled by using nested procedure calls to update them as data was being collected.

The changes made to the initial programming design used thousands less lines of code and reduced the number of variables.

Acknowledgments

This research is based, in part, on work that was supported by the U.S. Census Bureau. The authors of this paper and the programming staff would like to acknowledge the U.S. Census Bureau and the members of the BOC Survey of Health Insurance and Program Participation (SHIPP) team for their help and patience as the programming for the project developed. The views expressed here are those of the authors and not necessarily those of the U.S. Census Bureau.