THE DEVELOPMENT OF THE FAMILY RESOURCES SURVEY IN BLAISE FOR THE DEPARTMENT OF SOCIAL SECURITY, G.B.

Paddy Costigan, Sven Sjödin, Social and Community Planning Research, London, and Katarina Thomson, Joint Centre for Survey Methods, London

1. Introduction

This paper is based largely on the experience of Social and Community Planning (SCPR) in developing a Blaise questionnaire for a large and complex continuous survey, and on earlier work undertaken at SCPR and the Joint Centre for Survey Methods (JCSM) [1].

The start-up costs of acquiring expertise and laptops mean that simple surveys alone are unlikely to justify the expense of Computer Assisted Personal Interviewing (CAPI) in the near future despite the merits of 'clean data straight from the field'. Hence CAPI needs to prove its benefits for more complex surveys where extra constraints may come into play, such as hardware and software limitations, data storage, and the efficiency and maintainability of the program [2].

---

1) J. Martin and C. O'Muircheartaigh *Evaluation of Computer Assisted Survey Systems, Report 1: Introduction to Computer Assisted Survey Systems*, JCSM Working Paper Number 4, 1991.

2) A paper on this topic is also due to be delivered to the conference on Survey and Statistical Computing organised by the Study Group on Computers in Survey Analysis. The proceedings of that conference will be published in A. Westlake et al *Proceedings of the conference on Survey and Statistical Computing, Bristol, September, 24-26 1992*, Elsevier Science Publishers B.V., Amsterdam (forthcoming).

2. Key features and requirements of FRS

The Family Resources Survey (FRS) [3] is a complex survey: it is large and continuous (25,000 households per year); it deals with several hierarchical levels of data; it is a long interview (average 90 minutes per household); the co-operation of all adults in the household is required for the interview to be considered fully productive; and the interview collects large amounts of very detailed financial information, wherever possible verified by reference to source documents either in the interview or later.

3. Evaluation and choice of CAPI software

SCPR carried out an evaluation on behalf of the DSS of three leading CAPI software packages for potential use on the FRS. This was done by bench-testing them against a list of evaluation criteria. Blaise was chosen as the only package with sufficient potential for the FRS, principally because of the ease of movement around the questionnaire and the ability (using the *Table* facility) to display simultaneously, and move between, the answers of two adults being interviewed concurrently. There were, however, concerns over Blaise's capacity to deal with large, hierarchically structured questionnaires. Section 4.1 below discusses the solutions adopted.

4. Programming the questions

This section describes the problems encountered and solutions adopted in dealing with a number of major programming issues.

---

[3] The FRS is sponsored on behalf of the UK government by the Department of Social Security (DSS). The survey is due to be launched in October 1992, and will be carried out jointly by SCPR and the Office of Population Censuses and Surveys (OPCS).

4.1. Data hierarchies

The FRS contains complex data hierarchies: each household may contain one or several Benefit Units [4]; each Benefit Unit may contain one or two adults plus children; and each person (adult or child) may have several items of a particular sort, where information needs to be collected for each item, e.g. jobs, occupational pensions, financial holdings.

Because of the restrictions on using loops within loops in the ROUTE paragraph in Blaise version 2.4, it can be rather difficult to implement multiple levels of hierarchy. E.g. it is **not** possible to use the following pseudo-code to ask about three occupational pensions for each of ten persons in the household [5]:

```
FOR i := 1 TO 10 DO
   FOR j := 1 TO 3 DO
      Pension[i,j];
   ENDDO;
ENDDO;
```

Two levels of hierarchy can be created by using a loop within a table, but tables are not always the most suitable format. Two levels of hierarchy can also be created by defining questions as a block and declaring the block several times, but this is really only suitable if the inner loop is very small, e.g.:

---

4) A Benefit Unit generally consists of either a single person (with any dependent children) or a married or cohabiting couple (with any dependent children).

5) For clarity's sake, the question of whether there actually are ten people and whether they all have three pensions has been ignored in this example.

```
BLOCK PensionQuestions;
  .......
ENDBLOCK;

QUEST
  Pension1 : PensionQuestions;
  Pension2 : PensionQuestions;
  Pension3 : PensionQuestions;

ROUTE
  FOR i := 1 TO 10 DO
     Pension1;
     Pension2;
     Pension3;
  ENDDO;
```

By using tables, loops and re-declared questions in combination it is thus possible to create three levels of hierarchy. This is therefore not sufficient to deal with the FRS levels of Household-Benefit Unit-Person-Pension. A different approach was needed.

A related problem is run-time data storage space. The survey procedures allow for up to ten adults per household: the above design would require space for 30 instances of pensions for every household, whereas most households would probably only have one or two.

It soon became obvious that the questionnaire was in any case too large to be contained in one program. The original paper and pencil (PAPI) design had consisted of two questionnaires - one for the household and one for multiple adults. This was therefore a natural place to split the program.

A by-product of splitting the questionnaire was that the hierarchy problem was reduced. Thus the household program is run once for each household; a Benefit Unit program is run once for each Benefit Unit, with concurrent interviewing of its adult members. Up to nine Benefit Unit questionnaires can be linked to the household questionnaire. Thus within the Benefit Unit program, the upper layers of hierarchy can be ignored.

So, in the Benefit Unit program, the first level of hierarchy is whether you are dealing with Adult 1 or Adult 2 within the Benefit Unit. This is mostly handled via the table structure (see section 4.2 below on concurrent interviewing). The next level (e.g. multiple pensions) can be implemented using either a loop or a multiple declaration of questions, whichever is easier in the context (see section 4.3 below on different ways of implementing repeated questions). It would even be possible to implement a further level of hierarchy (e.g. some item of which there might be several per pension) by a combination of a loop and multiple declarations of questions.

This approach also reduces the data storage allocated: e.g. storage space for six pensions (i.e. three pensions each for two persons) is now required for all households. This is increased only if there are several Benefit Units in the household. This is clearly a substantial improvement on the figure of 30 given above.

The linking of the household and the individual programs is achieved via hierarchical serial numbers. The household program serial number contains both an address serial number and a household serial number (implemented as separate KEY questions) [6]. The Benefit Unit program has an additional KEY question: benefit unit serial number. Blaise reads information from the household questionnaire via an External Paragraph to several corresponding Benefit Unit questionnaires.

There are, of course, disadvantages in splitting the questionnaire into several programs. When the interviewer finishes one program, (s)he has to exit to an outer menu (currently implemented in Clipper) and then descend again for the new program. This becomes a particular problem when checks span questions in several programs. It is easy enough to implement such checks using information from the external paragraph, but they cannot take advantage of the usual Blaise facility whereby you can return to any question involved in the check simply by highlighting

---

[6] The sample is address-based and the sample-design must allow for multi-household addresses. The highest level of data analysis is, however, the household.

the question name. If the interviewer finds that (s)he has to alter an answer in an earlier program, (s)he has to swop programs and manually find the offending question. Also, structure checks to ensure that all relevant Benefit Unit questionnaires have been completed have to be implemented outside Blaise.

4.2. Concurrent interviewing

It was a requirement of the FRS that it should be possible to interview several adults concurrently. Ideally, this should be possible for all adults in the household, but in practice it has been limited to the two adults in each Benefit Unit.

There are several reasons for the requirement for concurrent interviewing. First, a number of questions deal with financial resources or income which may be held jointly by the adults in a Benefit Unit, e.g. savings accounts or the receipt of social security benefits. There is less risk of double-counting if both adults are interviewed jointly and checks can be programmed to prevent the double-claiming of benefits. Secondly, the interview is already very long (90 minutes per household). Concurrent interviewing involves some time saving (e.g. as some questions are read out only once). Thirdly, for the interview to be regarded as productive, all adults in the household must co-operate. There is less risk of 'losing' one respondent, if they are all interviewed together.

The model for the concurrent interviewing on the FRS is the Family Expenditure Survey, which has been run for many years by OPCS. In PAPI, the questions are printed down the left hand side of the page, with multiple answer columns for the adults. The interviewer has full discretion over the question order: (s)he can choose to ask each question of each adult in turn, or (s)he can ask a few questions of one person and then turn to the next person. The latter might be more sensible, say, if there is a series of questions all dealing with entries on the person's salary slip. If a person drops out, either temporarily or for the rest of the session, the interviewer can mark the spot and carry on with the other

respondent(s). This freedom has clear advantages in terms of interview dynamics, but leaves the door wide open for routing errors, especially as the FRS incorporates very complex routing.

The minimum requirements for the concurrent interviewing in CAPI were felt to be:

(a) that it should be possible to interview two persons (i.e. one Benefit Unit) at the same time;
(b) that it should be possible to run the interview with only one of the persons present; and
(c) that it should be possible to start the interview with two persons present, allow one person to drop out and continue the interview with the other person.

The first requirement - the ability to interview two persons at the same time - has been implemented using the *Table* structure. The Benefit Unit program thus consists mainly of a series of tables, each with two rows (one for each person), where the questions appear as columns.

In CAPI, the question order and where to switch between respondents has to be predetermined by the program designer. Where a sequence of questions seems to go together naturally, they have been put in the same table. Considerable pilot work has been done to establish how long to make the tables and where to place the breaks.

As for requirements (b) and (c), our approach - after a number of false starts - has been to intersperse the program with "Do you want to continue with Person x?" questions. If a person is 'suspended' at such a question, no further questions appear on the route for this person. The questions for the other person in the Benefit Unit are unaffected. If a respondent is not present at the start of the session, they can simply be 'suspended' at the outset and will be ignored for the rest of the session. If they drop out half way through, the interviewer will try to persuade them to stay until the next 'Suspend' question. If they insist on leaving between two 'Suspend' questions, the interviewer answers their questions using the 'Missing Information' key (see section 6.5) until the next 'Suspend' question. If a person reappears, and the interviewer wants to 'catch up'

with them, (s)he goes back to the question where they were first 'suspended' and changes it to 'continue'. Their questions will then reappear on the route. Once coded as 'continue', the answer to the a 'Suspend' question is protected, so the respondent's answers to subsequent questions remain displayed in all circumstances and their data cannot be lost.

4.3. Alternative strategies for repeated sets of questions

There are numerous places where the same questions are asked once for each of a number of items or persons. We would identify three main ways of presenting such sets of repeated questions. For example, the household questionnaire includes a number of questions about members of the household who need special help or attention. This could be presented in any of the following ways:

(a) An initial question finds out how many people in the household need special help. The program then repeats the sequence of subsidiary questions the requisite number of times. The sequence has to start with a question along the lines of "Who is the (first/second/third) person in the household who needs special help or attention?". (The word 'first', 'second', 'third' etc. can be inserted using text substitution.)

(b) An initial question finds out whether anyone in the household needs special help. If the answer is yes, the program begins the first iteration of the subsidiary questions. The sequence again has to start with a question along the lines of "Who is the (first/next) person in the household who needs special help or attention?". Either that question has an answer option 'No one else', or the sequence has to end with a question along the lines of "Is there anyone else in the household who needs special help or attention?".

(c) An initial multi-coded question establishes who, if anyone, needs special help. If one or more persons are coded at the multi-coded question, the subsidiary sequence is repeated for each person

affected. There is no need to ask each time who the person is (the correct name can be substituted into the question text) and there is no need to ask at the end of the sequence whether anyone else requires special help.

Our view is that, in general, (c) is the most elegant solution and the one that is easiest for the interviewers to deal with. Methods (a) and (b) could lead to confusion if there are a large number of items, and interviewers may end up duplicating or missing out information.

There are, however, situations where (a) or (b) might be preferable. When asking, for example, about motor vehicles available to the household, it seems more natural to use approach (a) and start by asking how many motor vehicles are involved. It is also perfectly natural to ask for each motor vehicle to whom it belongs.

Another consideration is data storage. If the program allows for up to ten adults and up to three cars per adult, storage would have to be allocated for 30 possible cars. If approach (b) or (c) was used, it might be thought sufficient to allow for a maximum of ten cars per household.

Finally, we have found it useful to implement some of the loops described above in the table format, so that the interviewer has an overview of the information given. This is particularly the case where the subsidiary sequence of questions is quite short and fits into one screen. If the sequence is longer (or contains large fields like open questions) so that the table scrolls sideways, the benefits of using a table are less apparent.

5. Checks and edits

One of the strengths of CAPI is that many checks normally done at the post-fieldwork edit stage can instead be done during the interview. Range errors can thus be eliminated and implausible information checked directly with the respondent. This improves data quality and reduces the editing time after fieldwork.

However, these benefits do not flow automatically from CAPI: they have to be worked at during the design stage. CAPI also introduces new checking requirements arising from the additional roles of the interviewer as key-punch operator and 'broker' of edit failures.

5.1. Strategies for minimising keying errors

Once interviewers have entered the data and it has been accepted, there is no independent record against which to check the answers. One hundred per cent verification of entries, i.e. the typing of each answer twice, can be used for a few critical questions, but is mostly impractical. For single-coded questions, the display of the short answer name next to the answer field is helpful in trying to cut keying errors. However, this does not operate within tables or in multi-coded or numeric questions. A new type of check is therefore needed to guard against interviewer keying errors.

For numeric data, soft checks on keying errors should be designed to be triggered whenever there is a real risk that the interviewer has keyed a zero too few or too many or has misplaced the decimal point, for example, to catch the interviewer who enters 500000 or 5000 instead of 50000. In some cases, it may be necessary to have fairly detailed information about the distribution of variables within the general population and important subgroups to design realistic checks. Non-numeric questions may also benefit from soft plausibility checks to catch both respondent and interviewer errors.

5.2. The design and storage of error messages

As discussed below, checks should be designed so as to be triggered only on rare occasions. One consequence is that the error messages will only rarely be seen. It is therefore essential that error messages are self-explanatory. The researcher needs carefully to think through and specify all error messages. It is important to remember that an 'error' can arise in at least three different ways: the respondent makes an error in answering;

the interviewer makes an error in keying the answer; or the respondent genuinely has circumstances which were not foreseen by the questionnaire designer. Error messages need to alert the interviewer to the problem and suggest reasons why it may have arisen without causing the interviewer to confront the respondent with the problem in an antagonistic way.

For some questions it is useful to store error messages as global text variables. This is appropriate when the same message applies to several checks, as it facilitates program maintenance and saves run-time storage space.

5.3. Practical limits on the number and types of checks

A necessary, but not sufficient, condition for 'clean data straight from the field' is that run-time edit checks can replace the entire office edit stage of a survey. We have already seen that CAPI introduces new requirements for checking. Checking requirements are therefore likely to be very heavy. There are, however, practical limits on the numbers and types of checks that can be implemented at run-time.

First, if the program triggers a hard check when there is no real error, the interview will come to a grinding halt. If the program frequently triggers soft checks when there is no real error, the interview becomes unwieldy and interviewers may start to ignore the error messages.

Secondly, checks implemented at run-time can really only deal with issues where the respondent can be expected to have the information. It is workable to implement a check that the amount received per week in Child Benefit should be made up of e.g. £9.65 plus multiples of £7.80, but not to implement a similar check on the amount of some benefit where entitlements vary depending on circumstances, and respondent and interviewer are unlikely to know the detailed eligibility rules.

Thirdly, there are situations where the run-time check has to be soft but clean data is nevertheless required, so a hard check needs to be imposed in the office. For example, a run-time check that only widows receive Widow's Benefit would have to be soft - you cannot take the risk of the interview grinding to a halt if the respondent insists that reality is otherwise. But it might be necessary to clean the data afterwards and enforce that only widows can receive Widow's Benefit.

Fourthly, we have found extensive checks to be costly in terms of program size and programming time.

Our experience is therefore that an office edit stage may still be necessary on a complex CAPI survey, although it should obviously be less time-consuming that the edit stage on a PAPI survey.

6. Significant limitations of Blaise version 2.4β

This section discusses some significant limitations that we have found in Blaise versions 2.37 and 2.4β. Although the limitations are often annoying and have sometimes constrained the way in which we have designed the program, none of them has proved to be an insuperable problem in the implementation of the FRS using Blaise.

6.1. Limitations on the depths of data hierarchies

The limitations imposed by Blaise on the depths of data hierarchies are potentially very serious for some surveys. The approaches used on the FRS to by-pass these limitations are discussed in section 4.1 above.

6.2. Size constraints

Even after splitting the FRS questionnaire into two programs (see section 4.1 above), we experienced a series of size constraints when working in Blaise version 2.37, mainly at compile and run time. These constraints

were so serious that we were at one point forced to split the questionnaire into three programs. However, the arrival of Blaise version 2.4β and the installation of DOS version 5 (which uses less RAM) on the laptops has alleviated these problems and enabled us to return to the two-way split of the questionnaire.

A lesser, but nevertheless annoying, size constraint is that the Blaise syntax checker is prepared to allow the assignment of much longer strings than the Turbo Pascal compiler. If very long strings have to be assigned, this should be done in two stages, e.g.

```
COMPUTE LongStr := 'This is the sort of long string that the Turbo
Pascal compiler ';
COMPUTE LongStr := LongStr + 'is not going to like at all. That is
so annoying!';
```

6.3. Numeric questions and variables

Although we were generally happy with the ability of Blaise to handle numeric data, we have found a few problems.

6.3.1. Rounding errors with real numbers

It is common practice in computing to recommend that equalities should never be used with real numbers, because of rounding errors when the numbers are stored in base 2 in the underlying hardware. Since this problem is common to all programming, it is not surprising that it should surface in Blaise. However, since Blaise is specifically designed to be user-friendly to non-programmers, the documentation should perhaps carry a warning about this.

Moreover, there is a particular problem where real numbers are used to define the range for a numeric question. If you define a range to be 0.00..99999.97, you would expect it to be possible to enter the number 99999.97. But this is not necessarily the case. Assigning the range 0.00..99999.00 to the question solves the problem.

There is a further rounding problem which arises when calculations are done using variables. Numeric questions may take real numbers of up to nine significant figures and hence up to eight decimal places, but variables used in calculations round to four decimal places.

6.3.2. Handling of 'Don't know' and 'Refusal' answers

Blaise assigns the next value above the top of the range for a numeric question to 'refusal' and the value after that to 'don't know'. If the answer is displayed in text substitution in a different question or used to perform a calculation, the result is nonsense values. To avoid this it is necessary either to ensure that all subsequent questions and checks which make use of the value are routed on the question being equal to RESPONSE or to filter the answer via a variable. The latter may not be suitable for checks, because the facility of jumping to the incorrect question by highlighting it, is available only where the check is specified on questions rather than on variables. In either case the solution requires a great deal of extra programming. We think that Blaise ought to be able to recognise its own don't know and refusal codes and have some set way of dealing with them in displays and calculations.

6.4. The apparent unpredictability of the triggering of checks and calculations

We have come across instances where the triggering of checks and calculations appeared to us to be unpredictable. We are not suggesting that there are bugs or mistakes in Blaise, merely that it is rather difficult to fathom the workings of the program and that this is not well documented.

One problem with calculations relates to the fact that Blaise initialises variables each time it checks a questionnaire. Say you have a questionnaire where you want to keep a record throughout the session of the value that a question had at the start of the session. This could be used, for example, to check that the interviewer does not change the serial

number. You might think that it would be possible to set up a variable (called, say, VSerNo) and initialise it at the start of the session with a statement along the lines of

```
IF VSerNo = 0 THEN
  COMPUTE VSerNo := SerNo
ENDIF;
```

But this does not work. Every time that SerNo is changed, VSerNo is also changed, because VSerNo is reset to 0 each time the questionnaire is checked. The solution to this problem is to use a PROTECT question and initialise it with the following statement:

```
CHECK
  IF (QSerNo = EMPTY) AND (SerNo <> EMPTY) THEN
     COMPUTE QSerNo := SerNo
  ENDIF;
```

Another problem appears to relate to checks tied to hidden questions. Sometimes checks that are hooked to hidden questions do not appear to be performed at the point where the hidden question appears in the ROUTE paragraph. We have not been able to isolate the precise conditions under which this occurs and we have often been put in the position of using trial and error, i.e. moving questions around within the ROUTE paragraph, until we hit upon a permutation that works. There is probably some very simple explanation to this, and we think this should be clearly explained in the documentation.

6.5. The lack of user defined keys

On the FRS, it is often the case that respondents do not have the information to hand at the time of the interview (e.g. their pay slip), but that they are willing to give the information to the interviewer later. The survey procedures allow for interviewers to recontact such respondents to collect these pieces of information. On a PAPI questionnaire, the interviewer would normally put a large star in the margin by such questions. We needed a way of emulating this in CAPI which would be available at all questions and easy to find afterwards, without cluttering

up screens and answer lists. We decided to use the refusal key (]). By amending the CAPITEXT file, we have caused the refusal key to display a series of exclamation marks in the answer field. However, this means that we now do not have a universal refusal key.

It would be useful to have a special user defined key similar to [and] which could be used for such purposes. It would also be useful to have the ability automatically to review afterwards those questions that have been answered with this special key.

7. Conclusion

This paper describes some of the issues which have arisen in the implementation of a large and complex survey in Blaise. In general, we are very happy with the Blaise software and feel that it is suitable for such a survey. However, some ingenuity has been required to implement features such as complex data hierarchies, concurrent interviewing, repeated sets of questions, and edit checks. We hope that other Blaise users can learn from our experience. We have also outlined what we consider to be some of the limitations of Blaise, particularly in relation to data hierarchies, size constraints, numeric questions and variables, the apparent unpredictability of the triggering of checks and calculations, and the lack of user-defined keys. We hope that these comments will be taken on board in future versions of Blaise.