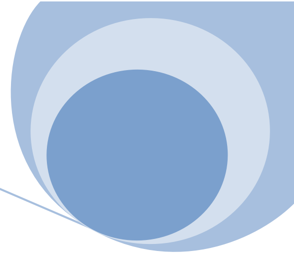
MRDCL

Excel Productivity Scripts (EPS)

Developing easily customizable systems to improve your productivity using MRDCL and Excel

Phil Hearn



Section 1: Introduction

One of the key productivity features within MRDCL is the set of tools that allow users to read Excel workbooks. This single feature means that a range of possibilities exist – from simply reading a list of items right through to developing systems that automate variable definitions and tables as well as merging and managing data. Scripts that utilise this feature are known as EPS (Excel Productivity Script).

This document is designed to introduce you to the tools that you have available with a short tutorial demonstrating the tools available. However, this document is more focused on how you can improve your productivity immediately and is provided with a number of working examples, which can be adapted to meet your specific needs.

The document should give you some good ideas as to how you can improve processes that you carry out every day. The focus here is on a little effort for a big pay back!

What is an EPS like?

An EPS is a combination of an Excel worksheet design coupled with a MRDCL script that can read and process it.

A good example of an EPS……

A good simple example of an EPS is the way that code lists are traditionally handled in market research. Typically, a coding team will develop code lists, a researcher will want sub-totals, leaving the MRDCL scriptwriter the task of co-ordinating this information. In other programs, this may require the scriptwriter to paste texts into scripts (text files) and to add syntactical commands to specify sub-total, ranking requirements, code groupings etc.

Using MRDCL, one worksheet can be used as a template to store the code list – this can be handled entirely by the coder. The researcher can add simple markers in the Excel spreadsheet to indicate which codes are used to make sub-totals. And, what does the MRDCL scriptwriter do? He/she only has to set a reference to the worksheet within the workbook using an EPS, so that everything is automated.

In other words, the coder does what they are responsible for; the researcher specifies what they need to; the scriptwriter only needs to set his MRDCL script to read the relevant worksheet. That, we believe, is efficiency in action.

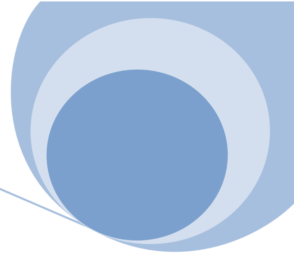
Why is an EPS important?

There are five key reasons why an EPS is important.

- **Everyone understands Excel** – Use of Excel is understood by almost everyone within the company. It is easy to train a junior member of staff with no expertise in MRDCL to enter or provide information in Excel. You can even get clients to provide information in a ready-to-use format that no needs NO further work to use.
- **Well designed systems can be re-used and shared** – a well designed EPS can be used across all or a number of projects and can even be re-used by different teams in different countries.
- **Changes to the worksheets do not require any updates for the script writer** – A well designed EPS will mean that the script writer merely sets a reference to the worksheet and uses the relevant driver file, setting the relevant parameters. It means that if a worksheet changes – for example, where codes are added, the scriptwriter does not have to make any changes at all.
- **Senior staff do not waste time doing junior tasks** – It is not uncommon for senior scriptwriters to spend a high proportion of their time – possibly as much as 90% of their time – doing simple work such as entering codes and texts. Systems designed using Excel mean that junior staff or coders can prepare much of an analysis specification.
- **MRDCL scriptwriters focus on scriptwriting tasks** – The greatest skill of a good MRDCL scriptwriter is usually their scriptwriting ability! This is not a surprise, of course, but much of a scriptwriter's time is often wasted changing code lists, sorting out data corrections and specifying simple tables. In other software packages, this I a problem. Why? Because, we believe, the software is not good enough. MRDCL allows juniors and other staff to do what they are good at while MRDCL scriptwriters focus on doing what few people understand.

The usual excuses for not using an EPS

- **They take longer to develop than entering the code** – This statement is absolutely true. However, it is highly short-sighted. Under pressure, it is easy to be short-sighted, of course, but companies making use of an EPS will get repaid for their initial effort many times over. As Excel is so easy to understand, minimal documentation of how to use a particular EPS is usually required as use is generally intuitive.
- **They are too difficult to develop** – Understanding how to develop an EPS can appear to be daunting at first, but it is surprising how easy it convert existing code so that it has a general purpose use as an EPS. Read the section on 'How to develop an EPS' or ask MRDC to provide you with a short online tutorial at a small fee, which will save you hours for years to come.
- **It means changing the way we have worked for years** – This is really no excuse. Whether you stick to the adage "There's no need to fix what isn't broke" or not – simply, if you can do something more efficiently, it has to be the right way to go.



Section 2: The working examples

This guide comes with 9 working examples, which are ready to use or to be adapted to meet your precise needs.

a) Reading lists

This section contains example scripts:

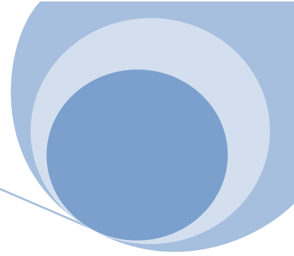
- Example 1: A simple EPS which reads a brand list from a spreadsheet. This example is used to teach you how to develop an EPS (see Section 3).
- Example 2: An example of how different languages can be read for texts
- Example 3: This EPS demonstrates how a list can be used to check data by specifying valid ranges for a series of fields
- Example 4: This is a full system for handling open ended code lists. It is developed in two stages to illustrate how additional features such as merging codes and applying ranking can be achieved.

b) Handling data

- Example 5: If you have data from another source which you wish to merge with your survey data, this is an ideal EPS for you.
- Example 6: This develops the previous example and processes a number of cluster solutions.
- Example 7: This shows a way of forcing data without having to physically change the initial data – other programs do not allow this forcing you to change the original source data. This provides you with an audit trail of your data changes, so that they can be easily amended.
- Example 8: This shows you how you might back code data from open ended questions in a slightly different way to Example 5.

c) Reading specifications

- Example 9: This template shows you how researchers can specify their tables in a simple form, so that most tables are automatically produced.



Section 3: How to develop an EPS

Example 1

The task: To develop an EPS that reads brand lists

The real advantage of an EPS is that the code can be re-used for different projects,

Developing an EPS can appear to be difficult. The recommended approach is to use reverse engineering techniques. In other words, write the code that reads one worksheet for one specific task and then make it general purpose. This can need a little more planning when you are trying to write options into your design, but this principle should simplify your task.

Let's start with a simple example where we want to read the codes and texts from a worksheet for a list of brands or makes/models. Taking the worksheet below, we might want to generate a variable definition with its texts.

code	Text
1	Airedale
2	Brassington
3	Cudworth
4	
5	Ellingham
6	Fry
7	
8	Harrison
9	
10	Jones
11	Kennedy
12	
13	Moore
14	North
15	Oswald
16	
17	
18	Redfern
19	Shuttersley
20	
21	Unicorn
22	
23	Williamson
24	
25	
26	Zebra

Assuming this is question 5 with the data in a 2-digit field in field 14-15, we could write the code for the information in this worksheet as follows:

```
ds $q5=$14-15/1,2,3,5,6,8,10,11,13,14,15,18,19,21,23,26,e,
x='Airedale;
Brassington;
Cudworth
…..
…..
Williamson;
Zebra;
Not stated <z>',
xt='Q5 Favourite brand',
```

However, you could write an EPS to handle this code. First of all, we need to open the workbook.

```
[*dbopen dbxls=myworkbook.xls]
```

Then, we choose to loop through the worksheet, starting at row 2 (the header is in row 1).

```
ds $q5=$14-15/[*dbloop db1=dbxls;thisworksheet][db1.code],[*dbend db1]e,
```

This means that MRDCL will pass through the worksheet *thisworksheet* and pick up all the codes in the column headed *code*. This is not quite right, though, as it will pick up every code. We, therefore, need to skip those codes which have no text in the column headed *text*. We can do this as follows:

```
ds $q5=$14-15/[*dbloop db1=dbxls;thisworksheet][*sk 99 string
db1.text.eq.''][db1.code], [*99][*dbend db1]e,
```

This will now list out the codes that we need - 1,2,3,5,6,8,10,11,13,14,15,18,19,21,23,26,

We can now add the x labels, such that the final code would look like this:

```
[*dbopen dbxls=..\myworkbook.xls]
ds $q5=$14-15/[*dbloop db1=dbxls;example1_worksheet][*sk 99 string
db1.text.eq.''][db1.code],[*99][*dbend db1]e,
x='
[*dbloop db1=dbxls;example1_worksheet]
[*sk 99 string db1.text.eq.'']
[db1.text];
[*99]
[*dbend db1]
Not stated<z>',
[*dbclose dbxls]
Xt='Q5 Favourite brand',
```

Now, we can take this one step further, a build a fully functioning EPS, which can be stored in a library and for use by many users. In this example, we may want to specify a different variable name, different data locations, a different title and a worksheet. So let's put those four parameters into a data statement.

```
[*data params=q5,14-15,Q5 Favourite brand,example1_worksheet]
```

By editing the above code, we can call these parameters.

```
[*data params=q5,14-15,Q5 Favourite brand]
ds $[params.1]=$[params.2]/[*dbloop db1=dbxls;[params.4]][*sk 99 string
db1.text.eq.''][db1.code],[*99][*dbend db1]e,
x='
[*dbloop db1=dbxls;example1_worksheet]
[*sk 99 string db1.text.eq.'']
[db1.text];
[*99]
[*dbend db1]
Not stated<z>',
xt='[params.3]',
```

Now, the final step. We can put the lines of code after the [*data statement into a separate file, so that it can be called with different parameters. For example:

```
[*data params=q5,14-15,Q5 Favourite brand,example1_worksheet]
[*insert example1.stp]
```

Example1.stp will contain the code below:

```
ds $[params.1]=$[params.2]/[*dbloop db1=dbxls;[params.4]][*sk 99 string
db1.text.eq.''][db1.code],[*99][*dbend db1]e,
x='
[*dbloop db1=dbxls;[params.4]]
[*sk 99 string db1.text.eq.'']
[db1.text];
[*99]
[*dbend db1]
Not stated<z>',
xt='[params.3]',
```

This is easy to do, you just need to substitute *[params.1]* for the variable name *q5*, *[params.2]* for the field *14-15*, *[params.3]* for the title *Q5 Favourite brand* and *[params.4]* for the worksheet *example1_worksheet*. The code now looks much more complex, but has been developed in logical steps.

We have now made a functioning EPS, which can be re-used within any number of projects and by different users. In practice, once the code in example1.stp is tested and correct, you would never change it again.

You can re-use it like this:

```
[*data params=q5,14-15,Q5 Favourite brand]
[*insert example1.stp]
[*data params=q6,16-17,Q5 Favourite brand]
[*insert example1.stp]
```

Note: example1.stp could be stored within the current directory or be used by a path. It may make sense to have a common path that many users share. For example:

```
[*insert z:\example1.stp]
```

The code for this example is provided in example1.

Section 4: The Examples

Introduction

This section contains a number of working examples, which explain the task, explain briefly how the code works and provides the working examples which you can adapt.

Each of the tasks aims to show the code necessary to handle the task.

Some of the examples have data to illustrate how they work and so that you can change them and re-test. Others can be seen by looking at the listing.txt which has the full code listed. Each example is driven by run.stp using the working book myworkbook.xls which must sit in a directory one level above for the examples to work.

Example 2

The task: To develop a system that would read different language code lists.

The approach: This is an extension of example 1, but allowing different translations of each text in different text columns. In addition to example 1, we need a setting or prompt which allows the user to choose the language required. This can be achieved through an ask command, a numeric pre-processor setting or a hard coded setting at the top of the main script file.

The solution: This uses a similar technique to example 1, but makes use of some other features. First of all, run.stp prompts the user for the language required. This is carried out by a data statement and an ask statement, which brings up a prompt. The two lines of code are:

```
[*data langs=uk,french,german]
[*ask langcode=langs]
```

MRDCL will set *langcode* to 1 if uk is chosen, 2 for French and 3 for german. The same effect could be achieved by using

```
pplangcode=<>,
```

If you put this into the control stage, the user will be prompted for a numeric value. A value of 1 would indicate uk, 2 for French etc. There are no prompts to remind the user.

Alternative, one could hard code the language by using:

```
[*set langcode=1]
```

The languages specified in this case match the headers for the text columns in the worksheet called *example2_worksheet*. In this case, *text_uk*, *text_french*, *text_german*.

For this solution, the title has been taken out of the *params data statement*; the three translations of the title are put into the *titles data statement*.

This solution will support more languages by extending the languages in the *langs data statement.*

Runs would crash if translations for the chosen language had not been specified. It would also support different code lists in different countries.

Example 3

The task: To develop a system that check valid price ranges for a list of products.

The approach: This just another application of the tools used in example 1. On this occasion, the worksheet contains the valid lower and upper prices are stored in the spreadsheet. This could be thousands of lines long and could contain many more fields. Indeed, it could be designed to store various sets of data.

The solution: Again, this is a question of deciding which things drive this series of checks. We need to set the field used to store the product code and the field used to store the price paid. These are the first two arguments of *data statement 500*. Finally, the name of the worksheet containing the valid price range is defined. This appears as follows:

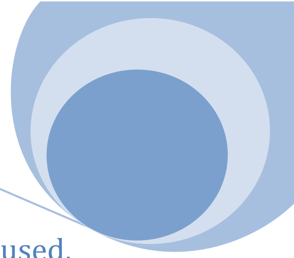
```
[*data 500=121-122,123-126,example3_worksheet]
[*insert example3.stp]

[*data 500=127-128,129-132,example3_worksheet]
[*insert example3.stp]
```

The file example3.stp is driven by the parameters in data 500. The code checks each row in the spreadsheet and makes a conditional check based on the product code. If there is no product text, no check is produced.

Again, this code can be re-used from project to project but more importantly, it can act as a master database, so that if new products are added, new checks will automatically be applied.

Example 4

The task: To develop a system that allows coders to build, edit and add to code frames and researchers to specify sub-totals that are required. The aim is that the script writer only has to set a reference to the worksheet and set some simple parameters. The system is also designed to automatically rank the sub-totals in descending order and the codes within that sub-total in descending order.



The approach: The design of the worksheet allows the user to specify the codes used, the sub-totals that are required and the specification of the codes which form those sub-totals.

The solution: It is important that the design of this EPS is easy for coders and researchers to use. There should be few rules. One of the few restrictions on this design (although it could be programmed differently) is that the sub-totals should be numbered 1 upwards in sequential order.

This EPS is designed to handle data that has been stored as a spreadfield. This means that the multiple responses are stored in a series of fields. For example, if up to 10 reasons may be coded, the data might appear as 2-digit codes in fields 11-12, 13-14, 15-16 etc up to 29-30. Alternatively, it may allow 4 'repeats' of 3 digit fields in 11-13, 14-16, 17-19 and 20-22. Of course, another solution could be developed to handle multi-coded binary data, data stored as 1's and 0's (using on/off bits) – the approach would be the same.

Looking at the design, the coder will assign codes using any numeric codes – in this case, the codes start at 1, but they could start at any number. As in example 1, empty texts are ignored. In the *group* column, a code is assigned to indicate which sub-total a code belongs to. Rows in the worksheet which are empty in the *code* column, but have text should have a marker 'x' in the *subtotal* column to indicate it is not a code, but a sub-total. Codes with nothing in the *sub-total* column are deemed not to part of any sub-total and are handled separately at the bottom of the resultant table.

This simple design is easily understood, although the code that runs this is quite complex. However, it was developed using the stages explained in Section 3. Once developed, there should never be a need to edit the code.

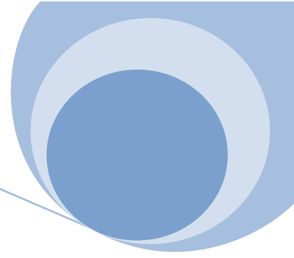
The EPS is called by 5 arguments – this is all that the scriptwriter would have to specify. The first of these arguments in *data statement cframes* is the name of the variable required, the second is the name of the worksheet which stores the code frame, the third is the title associated with the variable, the fourth is the data location of the first field, the fifth is the number of 'repeat' fields.

The code in example4.stp which is called, using the parameters in *data statement cframes*, is quite complex but runs in distinct stages.

Firstly, the variable is defined from the relevant data fields using the required number of repeats.

The next stage is to count the number of groups. This is achieved by looping through the worksheet and finding the highest group code.

Having found the highest group code, the EPS loops from 1 to the highest group code - 5, in the example provided. It then counts how codes form the 1st group. Having done this, it outputs the codes that comprise the sub-total followed by the individual codes falling into that group. It then repeats this for groups 2 through 5.



It counts the number of codes within a group first of all, so that it can control MRDCL code that needs to be output with + signs between each code.

It then follows a similar procedure for the texts. This time, it uses the sub-total's text for its label text and the individual code's texts. As processes this, there is some complex code that puts out all the MRDCL label controls to handle the two levels of ranking.

Finally, it outputs any codes which do not belong to a group; it also anchors them to the bottom of the table not using any ranking features.

To develop such an EPS and have it working correctly would take an experienced MRDCL user who has developed other EPS tools about 1 to 2 hours. Of course, the pay back is huge as the processing of code frames is now handled by the right people and is simple to amend. Changing sub-totals, for example, can be completed in a matter of seconds. Adding new codes also requires no work at all for the scriptwriter. A simple system has been developed to manage code frames for all projects.

Example 4a

This example extends this a little further by allowing the user to merge codes together before they are processed. For example, in example4a_worksheet, under the *merge* column, code 7 is set to be merged with code 6.

There now needs to be 6 parameters so that the start and finish position of the first field can be picked up. The EPS code then works out the width of the field and applies the merge for each code that it finds. Again, the solution is easy to use, but the code is fairly difficult use of MRDCL pre-processor code.

See also example 8/8a which shows how to develop this for a system that handles the actual coding of open ended texts.

Example 5

The task: To merge data from a spreadsheet with survey data.

The approach: There are two approaches, both of which are demonstrated giving the same result. The first approach is write code that will assign the relevant code for age, gender, income and ownership to each record (respid) by way of if statements. The second approach uses a technique which transfers a value to an integer. This second method is far quicker to process, but is limited to 9999 different assigned values (or respids, in this case). This same approach can be used to apply characteristics of respondents. For example, respondents interviewed in store 105 could have characteristics of that assigned – size of store, turnover of store, region, type of store etc.

The solution: The script begins by picking variables from the questionnaire. In this example, q1 is the only variable. This is followed by dummy definitions of variables and

their texts for data that is being merged from Excel. The *data statement fields* lists the fields which are to be read from the worksheet. In this case, the EPS matches the variable names to the column headers in the Excel worksheet – a different system could be developed. The statement **set countfields* counts the number of fields that are being merged from Excel. This enables the EPS to loop through the number of fields that are needed. No changes would be needed if more fields were added, except to list it in the *data statement fields*. Data is placed on to the dummy variables, using if statements. This is not particularly fast to process if there are several thousand of these statements.

The second approach is a little more complex, but processes marginally more quickly. Firstly, the respondent IDs are picked up from the *respids* column and stored in a variable. Note that the maximum length of a variable is 9999 bits. Therefore, this would need to be split if there were more than 9999 *respids* in the list. A counter called *countrespids* is set inside the dbloop which picks up the *respids*, so that we know the total number of *respids* having processed the loop.

As with the first approach, the number of fields is counted. An integer variable with an i_ prefix is assigned with the value for each *respid*. This is very efficient to process, although it requires an extra step to move the value from the i_ integer variable to the main variable. This can be performed by counting the bits in each variable listed in *data statement fields*, using the %items tool. Then, each value can be assigned, simply by looping for the number of valid codes.

This approach can be adapted for a number of purposes. It can be customised for more specific needs as demonstrated in Example 6.

Example 6

The task: To have a standard EPS that can be used to read multiple cluster solutions for any project.

The approach: The approach used here is similar to Example 5 except that it is tied more to a specific task. The EPS is driven by having one column headed as respid for the serial number. Then the cluster solutions are headed in line with the upper and lower limits of the cluster solutions used.

The solution: At the top of the script is a variable from the main data – gender, in this case. The solution is driven by the *data statement clusters*. It contains three arguments, the worksheet name and the lower and upper limits of the number of clusters in each solution. In this case, it starts at 4 and ends at 8.

The *data statement clusters* acts as a driver for the file pick_up_clusters.stp. In the same way as Example 5, the variable *respid* picks up all the serial numbers listed. The script then loops for the range of clusters required, picking up variables called i_cluster4 to i_cluster8. It then places them into variables cluster4 to cluster8, specifying the labels automatically.

This solution is simple to use and demonstrates how a system can be devised that anyone can use to produce clusters. This can obviously be applied to similar applications.

Example 7

The task: To have a system that logs changes to data that are required after data entry, keeping an audit trail that can be amended at any time. The system must also not physically change data, so that changes are not permanent. Data can be punched out to a new file if required.

The approach: The approach here is to lay out all the tasks that are needed so that MRDCL script can be written to read the EPS. The changes to the data are made after it is read in its raw form, so that the variables will be picking up clean data. These forces mean the data can re-checked/re-edited to test whether corrections have fixed data errors. Data corrections can be added to the bottom of the list, so that corrections work additively without damaging the original data. This is extremely important in situations where staff misunderstand instructions or clients issue contradictory instructions. The solution also allows additional fields to be entered so that the date of the correction and the person applying the change can be detailed.

The design of the Excel template to store data changes is easy to use. Where data from one column needs to be replaced, the *respid*, *column* and *settings* fields are completed. These are used to note the record number, the column to be changed and the new code respectively.

Where data from a field (range of columns) needs to be replaced, the *respid*, *field* and *settings* fields are completed. These are used to note the record number, the field to be changed and the new value respectively.

Where data from a field needs to be removed, the *respid* and *blank* fields are completed. These are used to note the record number and the field to be set to empty.

To use the template, the *respid* column can be used to list all records that need to have the same amendment. Where more than one record is selected, it must be separated by the + sign. A range of record numbers can be selected by using the form *x..y* where *x* is the lower value and *y* the higher value.

The columns for editor and date may be used to record the name of the person executing the change and the date. A drop down list can easily be set up in Excel for this purpose.

The solution: Again, this can be used routinely on every project. The EPS can be used for every project; the only thing that would change would be the data in the Excel worksheet.

The script first checks to see what type of edit is being implemented. Skips are in place where the columns *column, field or blank* are blank. If, for example, the column headed

column is not blank, it reads the data in the *respid* and *settings* column to drive out the data correction. The code works similarly for the columns headed *field* and *blank*.

The name of editor and the date are automatically output to the listing.txt file for easy viewing of changes.

Example 8

The task: To develop a system that allows back-coding of open ended questions.

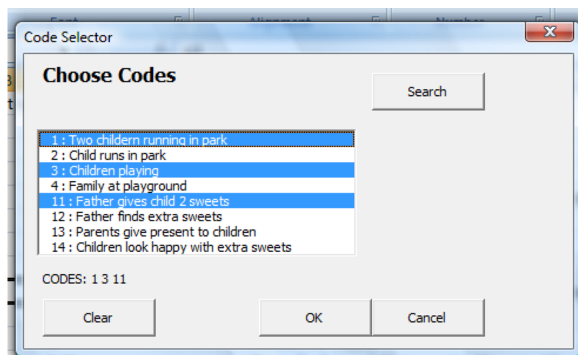
The approach: Coding from open ended questions can be merged with data without physically punching it on to the data or reading it from another ASCII or binary file. The recommended method is to code data into Excel worksheets, so that it can be merged using an EPS. Once again, this gives an audit trail of coding and can be used to record the coder and date etc.

The solution: The script file reads directly from the worksheet *example8_worksheet*. The questions are dummy-defined at the top of the script file. Sometimes, it is better to pick up the coded values in a temporary variable and then transfer them to the main variable as two steps. So, rather than using q1, q3 and q5, it may be better to use xq1a, xq3 and xq5, so that q1a, q3 and q5 are used to pick up the valid codes. This is shown in example 8a, which extends this example and uses macros in Excel to make coding and changes easier.

The script loops through the worksheet applying the codes to the questions as required. The system is easy for coders to use.

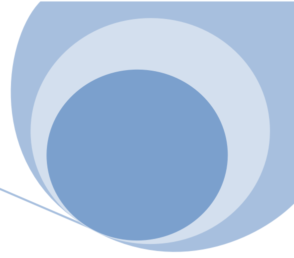
Oe.xls contain controls and macros to allow coders to code open ended data. The *control* worksheet handles the size and position of the main form that pops up. The worksheet *questions* lists the questions that may appear. These should match the worksheet names containing the code lists and used in the *coding* worksheet.

To pull up a list of possible codes or the codes already selected, you need to double click on the relevant row of the *coding* worksheet. For example, if you double click on cell B9 of the worksheet *coding*, a form will pop up with the codes already chosen (see below).



You can use control keys to select or de-select codes. The clear button will clear all codes selected, while the search button will allow you to search for text strings in a code list.

For fuller details on how to use this tool or to develop it further, please contact MRDC.



Example 8a

This solution can easily be extended by combining the EPS for example 8 with macros in Excel to improve usability.

The workbook oe.xls has macros running that detect user actions. For example, if a coder double clicks on any column other than the *codes* column of the worksheet *coding*, a form pops up with the code list, so that the coder can select the codes they want. The form also has a search facility so that text strings in long code lists can be found by intellisense text searching.

The workbook oe.xls is structured in a specific way and needs to be understood fully to use. The *control* worksheet determines where forms will pop up when activated by a double click on a worksheet. You can adjust the settings of the size and position of the form.

The *questions* worksheet lists the questions that may be coded. The *name* column defines the name of the question and the worksheet in which the code list will appear. Coders can use up to 9999 codes. Empty codes are ignored. Coding must take place in the *coding* worksheet. The headers for each of the columns determine the record number, the question being coded and the original text.

This type of tool can be extended to have other macros running and can be used in conjunction with example 4 which handles code lists for variables.

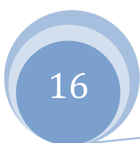
Example 9

The task: To develop a system that allows research executives to specify tables required in MRDCL. The aim here is that researchers specify as much as possible, leaving DP experts to specify more complex requirements.

The approach: A high percentage of tables in most projects are simple tables which cross one variable by a banner or another question. Sometimes filters are applied to these tables; sometimes summary tables are required. Researchers may also want to specify certain options, such as scoring a rating scale as +2 to -2.

The approach adopted here is to put all the standard requirements into a simple spreadsheet design that means that a researcher can provide a DP member of staff with a ready-to-use specification that will automatically run the basic tables. The system can be developed further to cover more sophisticated requirements, but starts with the basics. It also has an organised way for DP staff to slot in the more complex requirements without disturbing the worksheet that may control the bulk of table specifications.

The table specification template can be as easy or complex as you wish. It can be extended to handle more complex requirements, including difficult summary tables – this can all be achieved just by having a key word that tells the script file how to prepare the complex table.



The solution: This solution deals with basic cross tabulations, but also illustrates how to give options and deal with more complex requirements. The worksheet is largely self-explanatory. Each row of the worksheet is used to produce a table. Where complex requirements, the researcher may specify that a standard is not needed by putting 'n' in the *use column*. A written explanation of requirements can be entered in the *comments column*. In this case, the DP member of staff can enter the specifications in a file of their own name – in this case myq2.stp. This means that the template can remain as the master for the analysis and that a non-expert can all the simpler tables.

In this example, it is assumed that the variables for the rows and the banners have been pre-specified, but it would not be difficult to extend this so that the banners were also specified in another spreadsheet using easy-to-understand syntax.

The template allows for the specification of table titles, filters, filter texts and any of the 300+ options that exist within MRDCL. Some clients have made vast use of this tool, such that 90%+ of tables are specified this way. Again, if it is highly used, there is a benefit in having a macro that checks the logic of the syntax to make sure that there is nothing illogical.

The script file that reads this worksheet reads each row and makes a table applying the options selected as it loops through. To make it compile, dummy variables and banners have been specified. However, this could be extended to allow definitions of variables, banners and more complex tables.

Section 5: Other MRDCL information relating to developing an EPS

There are some important rules to follow when developing an EPS. You are advised to follow these rules:

- Always format each worksheet as a text. This is because the ODBC driver, that reads the sheet, will handle properly when it is formatted as text. Otherwise, it can read numerics wrongly from a syntactical viewpoint. For example, the figure 4 would be read as 4.0 which may not matter on all occasions, but could if it was referring to codes – in MRDCL syntax, 4.0 means code 4 and code 0. The best solution is to have a macro that automatically formats all new worksheets as text as they are generated. MRDC can help with this easily.
- Make sure you manage your EPS well. They may be shared by a number of users, so changing them after you have deployed them may cause other users problems. It usually better to make a new EPS or new version, so that users can choose to upgrade if they wish without damaging old script files.
- Consider writing macros to check the contents of worksheets. This can save time with finding errors in MRDCL code and will mean that users can work without having to worry about making errors.
- You will not encounter any problems with limits. Many users have developed complex systems that once developed provide huge benefits. The whole idea of an EPS is to save time in the long run so good design is all important.
- Make sure usage of an EPS is easy to understand – particularly for infrequent users. Consider having a help page so that users can use the spreadsheet design without any training and minimal explanation.
- Put someone in charge of the series of EPS files you generate. That person will be responsible for publishing new ones, providing information to others and helping to make the most of the EPS files within your organisation.
- Ask for help if you need to. At MRDC, we are pleased to help you design systems that meet your precise needs. We have been commissioned to provide a number of EPS files for customers, ranging from quite simple to highly complex. The cost is usually quite small (or even free!) – regardless, it will be small price to pay for saving hours of time.
- MRDC also runs both face to face and online training sessions to help users understand the right approaches and how to develop this type of script code. There are 3 levels – beginner, intermediate and advanced. These are available at competitive prices and can be conducted online