Customized Markdown and `.docx` tables using `listtab` and `docxtab`

Roger B. Newson
roger.newson@kcl.ac.uk
http://www.rogernewsonresources.org.uk

Cancer Prevention Group, School of Cancer & Pharmaceutical Sciences, King's College London

Presented at the 2023 UK Stata Conference, London,
7–8 September, 2023
Downloadable from the conference website at
https://econpapers.repec.org/paper/boclsug23/

Stata has *always* had tables

- They are called **Stata datasets**, can live in files or in frames[1], and are **relational tables** in the sense of Date (2003)[2], with tuples (or rows) called **observations**.
- And they have a **primary key** defined by the virtual variable `_n`, *preferably* preceded by the `sortedby` *`varlist`*.
- And their variables can be listed, using the SSC package `listtab`[3], to **tables in documents** in an *endless* variety of formats, specified by the **row style option** `rstyle()`.
- Each row style is defined as a combination of a `begin(`*`string`*`)`, a `delimiter(`*`string`*`)`, and an `end(`*`string`*`)`.
- *However*, today we will be using `rstyle(markdown)`, defined *simply* as `begin(|) delimiter(|) end(|)`, which can be used for making HTML tables (via `markdown`).
- And now we can also output variables to tables in Open Office XML (`.docx`) documents, using `putdocx` and the SSC package `docxtab`.

Stata has *always* had tables

- They are called **Stata datasets**, can live in files or in frames[1], and are **relational tables** in the sense of Date (2003)[2], with tuples (or rows) called **observations**.
- And they have a **primary key** defined by the virtual variable `_n`, *preferably* preceded by the `sortedby` *`varlist`*.
- And their variables can be listed, using the SSC package `listtab`[3], to **tables in documents** in an *endless* variety of formats, specified by the **row style option** `rstyle()`.
- Each row style is defined as a combination of a `begin(`*`string`*`)`, a `delimiter(`*`string`*`)`, and an `end(`*`string`*`)`.
- *However*, today we will be using `rstyle(markdown)`, defined *simply* as `begin(|) delimiter(|) end(|)`, which can be used for making HTML tables (via `markdown`).
- And now we can also output variables to tables in Open Office XML (`.docx`) documents, using `putdocx` and the SSC package `docxtab`.

Stata has *always* had tables

- They are called **Stata datasets**, can live in files or in frames[1], and are **relational tables** in the sense of Date (2003)[2], with tuples (or rows) called **observations**.
- And they have a **primary key** defined by the virtual variable `_n`, *preferably* preceded by the `sortedby` *`varlist`*.
- And their variables can be listed, using the SSC package `listtab`[3], to **tables in documents** in an *endless* variety of formats, specified by the **row style option** `rstyle()`.
- Each row style is defined as a combination of a `begin(`*`string`*`)`, a `delimiter(`*`string`*`)`, and an `end(`*`string`*`)`.
- *However*, today we will be using `rstyle(markdown)`, defined *simply* as `begin(|) delimiter(|) end(|)`, which can be used for making HTML tables (via `markdown`).
- And now we can also output variables to tables in Open Office XML (`.docx`) documents, using `putdocx` and the SSC package `docxtab`.

Stata has *always* had tables

- They are called **Stata datasets**, can live in files or in frames[1], and are **relational tables** in the sense of Date (2003)[2], with tuples (or rows) called **observations**.
- And they have a **primary key** defined by the virtual variable `_n`, *preferably* preceded by the `sortedby` *`varlist`*.
- And their variables can be listed, using the SSC package `listtab`[3], to **tables in documents** in an *endless* variety of formats, specified by the **row style option** `rstyle()`.
- Each row style is defined as a combination of a `begin(`*`string`*`)`, a `delimiter(`*`string`*`)`, and an `end(`*`string`*`)`.
- *However*, today we will be using `rstyle(markdown)`, defined *simply* as `begin(|) delimiter(|) end(|)`, which can be used for making HTML tables (via `markdown`).
- And now we can also output variables to tables in Open Office XML (`.docx`) documents, using `putdocx` and the SSC package `docxtab`.

Stata has *always* had tables

- They are called **Stata datasets**, can live in files or in frames[1], and are **relational tables** in the sense of Date (2003)[2], with tuples (or rows) called **observations**.
- And they have a **primary key** defined by the virtual variable `_n`, *preferably* preceded by the `sortedby` *`varlist`*.
- And their variables can be listed, using the SSC package `listtab`[3], to **tables in documents** in an *endless* variety of formats, specified by the **row style option** `rstyle()`.
- Each row style is defined as a combination of a `begin(`*`string`*`)`, a `delimiter(`*`string`*`)`, and an `end(`*`string`*`)`.
- *However*, today we will be using `rstyle(markdown)`, defined *simply* as `begin(|) delimiter(|) end(|)`, which can be used for making HTML tables (via `markdown`).
- And now we can also output variables to tables in Open Office XML (`.docx`) documents, using `putdocx` and the SSC package `docxtab`.

Stata has *always* had tables

- They are called **Stata datasets**, can live in files or in frames[1], and are **relational tables** in the sense of Date (2003)[2], with tuples (or rows) called **observations**.
- And they have a **primary key** defined by the virtual variable `_n`, *preferably* preceded by the `sortedby` *`varlist`*.
- And their variables can be listed, using the SSC package `listtab`[3], to **tables in documents** in an *endless* variety of formats, specified by the **row style option** `rstyle()`.
- Each row style is defined as a combination of a `begin(`*`string`*`)`, a `delimiter(`*`string`*`)`, and an `end(`*`string`*`)`.
- *However*, today we will be using `rstyle(markdown)`, defined *simply* as `begin(|) delimiter(|) end(|)`, which can be used for making HTML tables (via `markdown`).
- And now we can also output variables to tables in Open Office XML (`.docx`) documents, using `putdocx` and the SSC package `docxtab`.

Stata has *always* had tables

- They are called **Stata datasets**, can live in files or in frames[1], and are **relational tables** in the sense of Date (2003)[2], with tuples (or rows) called **observations**.
- And they have a **primary key** defined by the virtual variable `_n`, *preferably* preceded by the `sortedby` *`varlist`*.
- And their variables can be listed, using the SSC package `listtab`[3], to **tables in documents** in an *endless* variety of formats, specified by the **row style option** `rstyle()`.
- Each row style is defined as a combination of a `begin(`*`string`*`)`, a `delimiter(`*`string`*`)`, and an `end(`*`string`*`)`.
- *However*, today we will be using `rstyle(markdown)`, defined *simply* as `begin(|) delimiter(|) end(|)`, which can be used for making HTML tables (via `markdown`).
- And now we can also output variables to tables in Open Office XML (`.docx`) documents, using `putdocx` and the SSC package `docxtab`.

Recommendations for datasets output using `listtab`

The `listtab` package inputs a *`varlist`* of variables for output. The definitive SJ paper on `listtab`[3] gives some general recommendations about these input variables:

1. If numeric, they should be converted to **string variables**, because numeric variables in tables in documents can have formats and/or fonts and/or parentheses and/or *P*–value stars.
2. The first of these input variables should be a **row label variable**.
3. And all of these input variables should have a list of **variable characteristics**, specifying one or more rows of **column labels**, and possibly other column attributes (like alignment).

Note that it is also a good idea for the input dataset to have a **primary key** of variables *not* in the *`varlist`* input by `listtab`, and to calculate the row label variable from these. And it is also a good idea for the above destructive process to take place between a `preserve` and a `restore`. The recommended sequence is called the **DCRIL path** (decode, characterize, reshape, insert, list).

We will assume that we have a resultsset. . .

In the extended `auto` dataset created by the SSC package `xauto`, we use our old friend the `parmby` module of the SSC package `parmest` to fit 2 regression models of fuel consumption (in nipperkins per mile) with respect to car weight (in US tons), one for non–US models and one for US models. We save the results in a resultsset in memory, with 1 observation per car model origin per model parameter, overwriting the original dataset. The code to do this is as follows:

```
parmby "regress npm tons, vce(robust)",
  fast by(us) label
  escal(N) rename(es_1 N)
  format(estimate min* max* %8.3f p %-8.3e);
```

Note that the covariate variable label is saved in a string variable `label`, and the number of car models for each car origin group is saved in a variable `es_1`, renamed to `N`.

. . .and we seem to have a resultsset. . .

When we `describe` this resultsset, we find these variables:

```
. describe, full;

Contains data
 Observations:             4
    Variables:            12
-------------------------------------------------------------------------------
Variable      Storage   Display    Value
    name         type    format    label      Variable label
-------------------------------------------------------------------------------
us              byte    %8.0g      us         US or non-US model
parmseq         byte    %12.0g                Parameter sequence number
parm            str5    %9s                   Parameter name
label           str16   %16s                  Parameter label
estimate        double  %8.3f                 Parameter estimate
stderr          double  %10.0g                SE of parameter estimate
dof             byte    %10.0g                Degrees of freedom
t               double  %10.0g                t-test statistic
p               double  %-8.3e                P-value
min95           double  %8.3f                 Lower 95% confidence limit
max95           double  %8.3f                 Upper 95% confidence limit
N               byte    %10.0g                e(N)
-------------------------------------------------------------------------------
Sorted by: us  parmseq
     Note: Dataset has changed since last saved.
```

It is keyed by the car origin variable `us` and the parameter sequence number variable `parmseq`.

. . .but it seems to be a *complicated* resultsset!

When we `list` *some* of these variables, we find that there are 3 separate variables, `parmseq`, `label`, and `parm`, containing the parameter sequence number, covariate label, and covariate name, respectively. And the sample number `N` is repeated:

```
. by us: list parmseq label parm N estimate min* max* p, noobs;

-------------------------------------------------------------------------------
-> us = Non-US

  +-------------------------------------------------------------------------+
  | parmseq              label    parm    N   estimate    min95    max95   p |
  |-------------------------------------------------------------------------|
  |       1   Weight (US tons)    tons   22     11.059    8.130   13.987   1.5e-07 |
  |       2           Constant   _cons   22     -1.764   -5.382    1.853   3.2e-01 |
  +-------------------------------------------------------------------------+

-------------------------------------------------------------------------------
-> us = US

  +-------------------------------------------------------------------------+
  | parmseq              label    parm    N   estimate    min95   max95   p |
  |-------------------------------------------------------------------------|
  |       1   Weight (US tons)    tons   52      7.885    6.484   9.286   2.2e-15 |
  |       2           Constant   _cons   52      0.537   -1.549   2.622   6.1e-01 |
  +-------------------------------------------------------------------------+
```

So can we simplify our resultsset?

The central role of `sdecode` (and `sencode`) in resultsprocessing

- The SSC package `sdecode` is a "super–decoder", combining the roles of `decode` and `tostring`.
- It has a `replace` option, so the output string variable can replace the input numeric variable (inheriting its name, variable label, and position).
- It has options `prefix(`*`string`*`)` and `suffix(`*`string`*`)`, enabling us to add prefixes or suffixes, containing parentheses and/or *P*–value stars and/or font specifications.
- It has an option `esub(`*`substitution_rule`*`)`, enabling us to substitute Markdown, HTML, TEX, or RTF superscripting in e–formatted variables.
- And it has a number of **dependent SSC packages**, like `bmjcip` and `insingap`.
- The SSC package `sencode`[4] is a "super–encoder", with options `replace` and `gsort()` (for specifying coding orders using sort–key variables).

The central role of `sdecode` (and `sencode`) in resultsprocessing

- The SSC package `sdecode` is a "super–decoder", combining the roles of `decode` and `tostring`.
- It has a `replace` option, so the output string variable can replace the input numeric variable (inheriting its name, variable label, and position).
- It has options `prefix(`*`string`*`)` and `suffix(`*`string`*`)`, enabling us to add prefixes or suffixes, containing parentheses and/or *P*–value stars and/or font specifications.
- It has an option `esub(`*`substitution_rule`*`)`, enabling us to substitute Markdown, HTML, TEX, or RTF superscripting in e–formatted variables.
- And it has a number of **dependent SSC packages**, like `bmjcip` and `insingap`.
- The SSC package `sencode`[4] is a "super–encoder", with options `replace` and `gsort()` (for specifying coding orders using sort–key variables).

The central role of `sdecode` (and `sencode`) in resultsprocessing

- The SSC package `sdecode` is a "super–decoder", combining the roles of `decode` and `tostring`.
- It has a `replace` option, so the output string variable can replace the input numeric variable (inheriting its name, variable label, and position).
- It has options `prefix(`*`string`*`)` and `suffix(`*`string`*`)`, enabling us to add prefixes or suffixes, containing parentheses and/or *P*–value stars and/or font specifications.
- It has an option `esub(`*`substitution_rule`*`)`, enabling us to substitute Markdown, HTML, TEX, or RTF superscripting in e–formatted variables.
- And it has a number of **dependent SSC packages**, like `bmjcip` and `insingap`.
- The SSC package `sencode`[4] is a "super–encoder", with options `replace` and `gsort()` (for specifying coding orders using sort–key variables).

The central role of `sdecode` (and `sencode`) in resultsprocessing

- The SSC package `sdecode` is a "super–decoder", combining the roles of `decode` and `tostring`.
- It has a `replace` option, so the output string variable can replace the input numeric variable (inheriting its name, variable label, and position).
- It has options `prefix(`*`string`*`)` and `suffix(`*`string`*`)`, enabling us to add prefixes or suffixes, containing parentheses and/or *P*–value stars and/or font specifications.
- It has an option `esub(`*`substitution_rule`*`)`, enabling us to substitute Markdown, HTML, TEX, or RTF superscripting in e–formatted variables.
- And it has a number of **dependent SSC packages**, like `bmjcip` and `insingap`.
- The SSC package `sencode`[4] is a "super–encoder", with options `replace` and `gsort()` (for specifying coding orders using sort–key variables).

The central role of `sdecode` (and `sencode`) in resultsprocessing

- The SSC package `sdecode` is a "super–decoder", combining the roles of `decode` and `tostring`.
- It has a `replace` option, so the output string variable can replace the input numeric variable (inheriting its name, variable label, and position).
- It has options `prefix(`*`string`*`)` and `suffix(`*`string`*`)`, enabling us to add prefixes or suffixes, containing parentheses and/or *P*–value stars and/or font specifications.
- It has an option `esub(`*`substitution_rule`*`)`, enabling us to substitute Markdown, HTML, TEX, or RTF superscripting in e–formatted variables.
- And it has a number of **dependent SSC packages**, like `bmjcip` and `insingap`.
- The SSC package `sencode`[4] is a "super–encoder", with options `replace` and `gsort()` (for specifying coding orders using sort–key variables).

The central role of `sdecode` (and `sencode`) in resultsprocessing

- The SSC package `sdecode` is a "super–decoder", combining the roles of `decode` and `tostring`.
- It has a `replace` option, so the output string variable can replace the input numeric variable (inheriting its name, variable label, and position).
- It has options `prefix(`*`string`*`)` and `suffix(`*`string`*`)`, enabling us to add prefixes or suffixes, containing parentheses and/or *P*–value stars and/or font specifications.
- It has an option `esub(`*`substitution_rule`*`)`, enabling us to substitute Markdown, HTML, TEX, or RTF superscripting in e–formatted variables.
- And it has a number of **dependent SSC packages**, like `bmjcip` and `insingap`.
- The SSC package `sencode`[4] is a "super–encoder", with options `replace` and `gsort()` (for specifying coding orders using sort–key variables).

The central role of `sdecode` (and `sencode`) in resultsprocessing

- The SSC package `sdecode` is a "super–decoder", combining the roles of `decode` and `tostring`.
- It has a `replace` option, so the output string variable can replace the input numeric variable (inheriting its name, variable label, and position).
- It has options `prefix(`*`string`*`)` and `suffix(`*`string`*`)`, enabling us to add prefixes or suffixes, containing parentheses and/or *P*–value stars and/or font specifications.
- It has an option `esub(`*`substitution_rule`*`)`, enabling us to substitute Markdown, HTML, TEX, or RTF superscripting in e–formatted variables.
- And it has a number of **dependent SSC packages**, like `bmjcip` and `insingap`.
- The SSC package `sencode`[4] is a "super–encoder", with options `replace` and `gsort()` (for specifying coding orders using sort–key variables).

Using `sencode` to create an improved model–parameter variable

In our resultsset, we demonstrate `sencode` by creating a new model–parameter variable `modparam`, by encoding the parameter label variable `label` in the order of the parameter sequence variable `parmseq`:

```
sencode label, gsort(parmseq) gene(modparam);
lab var modparam "Model parameter";
drop parmseq parm label;
keyby us modparam;
```

Note that we can `drop` the old parameter variables `parmseq`, `parm`, and `label`, and use the SSC package `keyby` to sort the dataset by the car origin group variable `us` and the model–parameter variable `modparam`, checking that the 2 key variables uniquely identify the observations.

Using sdecode and sencode to create an improved car origin variable

In our resultsset, we sdecode the car origin variable us to a string variable us2, use replace to add to us2 the corresponding value (in parenthesis) of the sample–number variable N, and sencode us2 back to numeric, in the order of us:

```
sdecode us, gene(us2);
replace us2 = us2 + " (N=" + string(N) + ")";
sencode us2, gsort(us) replace;
keep us2 modparam estimate min* max* p;
keyby us2 modparam;
```

Note that we can now keep only a short list of really necessary variables, and use keyby to key this reduced dataset by us2 and modparam.

And here we have our new slimmer resultsset

Using `list`, we view our new resultsset, starting with the key variables `us2` and `modparam`, with 1 observation per car–origin group per model parameter:

```
. list us2 modparam estimate min* max* p, noobs;

  +-------------------------------------------------------------------------+
  |           us2           modparam   estimate    min95    max95   p      |
  |-------------------------------------------------------------------------|
  | Non-US (N=22)   Weight (US tons)     11.059    8.130   13.987   1.5e-07 |
  | Non-US (N=22)            Constant     -1.764   -5.382    1.853   3.2e-01 |
  |     US (N=52)   Weight (US tons)      7.885    6.484    9.286   2.2e-15 |
  |     US (N=52)            Constant      0.537   -1.549    2.622   6.1e-01 |
  +-------------------------------------------------------------------------+
```

We can now `save` our resultsset (to `result1.dta`), and `use` it to generate Markdown, HTML, and `docx` resultstables.

Creating Markdown tables using the SSC package `listtab`

- A **Markdown language** is a shorthand language for HTML.
- There are many Markdowns, but we will be using **Stata Markdown**, which allows us to drop into HTML, and to make tables.
- In Stata, Markdown documents can be written using the `file` utility, beginning with `file open` and ending with `file close`, preferably with a `capture noisily` group[5] in between, in case there are errors.
- And, in these Markdown documents, each table can be made using `listtab`, usually between a `preserve` and a `restore`.
- And, when the Markdown document has been written, we can convert it to browser–ready HTML, using the `markdown` command.
- We will demonstrate this method for creating a HTML document, containing resultstables from our resultsset.

Creating Markdown tables using the SSC package `listtab`

- A **Markdown language** is a shorthand language for HTML.
- There are many Markdowns, but we will be using **Stata Markdown**, which allows us to drop into HTML, and to make tables.
- In Stata, Markdown documents can be written using the `file` utility, beginning with `file open` and ending with `file close`, preferably with a `capture noisily` group[5] in between, in case there are errors.
- And, in these Markdown documents, each table can be made using `listtab`, usually between a `preserve` and a `restore`.
- And, when the Markdown document has been written, we can convert it to browser–ready HTML, using the `markdown` command.
- We will demonstrate this method for creating a HTML document, containing resultstables from our resultsset.

Creating Markdown tables using the SSC package `listtab`

- A **Markdown language** is a shorthand language for HTML.
- There are many Markdowns, but we will be using **Stata Markdown**, which allows us to drop into HTML, and to make tables.
- In Stata, Markdown documents can be written using the `file` utility, beginning with `file open` and ending with `file close`, preferably with a `capture noisily` group[5] in between, in case there are errors.
- And, in these Markdown documents, each table can be made using `listtab`, usually between a `preserve` and a `restore`.
- And, when the Markdown document has been written, we can convert it to browser–ready HTML, using the `markdown` command.
- We will demonstrate this method for creating a HTML document, containing resultstables from our resultsset.

Creating Markdown tables using the SSC package `listtab`

- A **Markdown language** is a shorthand language for HTML.
- There are many Markdowns, but we will be using **Stata Markdown**, which allows us to drop into HTML, and to make tables.
- In Stata, Markdown documents can be written using the `file` utility, beginning with `file open` and ending with `file close`, preferably with a `capture noisily` group[5] in between, in case there are errors.
- And, in these Markdown documents, each table can be made using `listtab`, usually between a `preserve` and a `restore`.
- And, when the Markdown document has been written, we can convert it to browser–ready HTML, using the `markdown` command.
- We will demonstrate this method for creating a HTML document, containing resultstables from our resultsset.

Creating Markdown tables using the SSC package `listtab`

- A **Markdown language** is a shorthand language for HTML.
- There are many Markdowns, but we will be using **Stata Markdown**, which allows us to drop into HTML, and to make tables.
- In Stata, Markdown documents can be written using the `file` utility, beginning with `file open` and ending with `file close`, preferably with a `capture noisily` group[5] in between, in case there are errors.
- And, in these Markdown documents, each table can be made using `listtab`, usually between a `preserve` and a `restore`.
- And, when the Markdown document has been written, we can convert it to browser–ready HTML, using the `markdown` command.
- We will demonstrate this method for creating a HTML document, containing resultstables from our resultsset.

Creating Markdown tables using the SSC package `listtab`

- A **Markdown language** is a shorthand language for HTML.
- There are many Markdowns, but we will be using **Stata Markdown**, which allows us to drop into HTML, and to make tables.
- In Stata, Markdown documents can be written using the `file` utility, beginning with `file open` and ending with `file close`, preferably with a `capture noisily` group[5] in between, in case there are errors.
- And, in these Markdown documents, each table can be made using `listtab`, usually between a `preserve` and a `restore`.
- And, when the Markdown document has been written, we can convert it to browser–ready HTML, using the `markdown` command.
- We will demonstrate this method for creating a HTML document, containing resultstables from our resultsset.

Creating Markdown tables using the SSC package `listtab`

- A **Markdown language** is a shorthand language for HTML.
- There are many Markdowns, but we will be using **Stata Markdown**, which allows us to drop into HTML, and to make tables.
- In Stata, Markdown documents can be written using the `file` utility, beginning with `file open` and ending with `file close`, preferably with a `capture noisily` group[5] in between, in case there are errors.
- And, in these Markdown documents, each table can be made using `listtab`, usually between a `preserve` and a `restore`.
- And, when the Markdown document has been written, we can convert it to browser–ready HTML, using the `markdown` command.
- We will demonstrate this method for creating a HTML document, containing resultstables from our resultsset.

Creating a Markdown document from our resultsset

- The do–file `mymddoc1.do` generates a HTML document `mymddoc1.htm` (via a Markdown document `mymddoc1.md`) from our resultsset in `result1.dta`.
- We use the SSC package `tfinsert` to write the text, and then create 2 alternative tables from our resultsset using `listtab`, each between a `preserve` and a `restore`.
- For each table, we start by using the SSC packages `sdecode` to decode the model–parameter variable `modparam` to a row–label variable `rowlab`, and `bmjcip` to decode the estimates, confidence limits, and *P*–values.
- We then use the SSC package `chardef` to set **column characteristics**, specifying the column labels and alignments.
- In the first table, we then use the SSC package `insingap` to insert a **gap row** at the start of each car–origin by–group.
- And, in the second table, we then use the SSC package `xrewide` (an extension of `reshape wide`) to reshape the table to give the 2 car–origin groups side by side.

Creating a Markdown document from our resultsset

- The do–file `mymddoc1.do` generates a HTML document `mymddoc1.htm` (via a Markdown document `mymddoc1.md`) from our resultsset in `result1.dta`.
- We use the SSC package `tfinsert` to write the text, and then create 2 alternative tables from our resultsset using `listtab`, each between a `preserve` and a `restore`.
- For each table, we start by using the SSC packages `sdecode` to decode the model–parameter variable `modparam` to a row–label variable `rowlab`, and `bmjcip` to decode the estimates, confidence limits, and *P*–values.
- We then use the SSC package `chardef` to set **column characteristics**, specifying the column labels and alignments.
- In the first table, we then use the SSC package `insingap` to insert a **gap row** at the start of each car–origin by–group.
- And, in the second table, we then use the SSC package `xrewide` (an extension of `reshape wide`) to reshape the table to give the 2 car–origin groups side by side.

Creating a Markdown document from our resultsset

- The do–file `mymddoc1.do` generates a HTML document `mymddoc1.htm` (via a Markdown document `mymddoc1.md`) from our resultsset in `result1.dta`.
- We use the SSC package `tfinsert` to write the text, and then create 2 alternative tables from our resultsset using `listtab`, each between a `preserve` and a `restore`.
- For each table, we start by using the SSC packages `sdecode` to decode the model–parameter variable `modparam` to a row–label variable `rowlab`, and `bmjcip` to decode the estimates, confidence limits, and *P*–values.
- We then use the SSC package `chardef` to set **column characteristics**, specifying the column labels and alignments.
- In the first table, we then use the SSC package `insingap` to insert a **gap row** at the start of each car–origin by–group.
- And, in the second table, we then use the SSC package `xrewide` (an extension of `reshape wide`) to reshape the table to give the 2 car–origin groups side by side.

Creating a Markdown document from our resultsset

- The do–file `mymddoc1.do` generates a HTML document `mymddoc1.htm` (via a Markdown document `mymddoc1.md`) from our resultsset in `result1.dta`.
- We use the SSC package `tfinsert` to write the text, and then create 2 alternative tables from our resultsset using `listtab`, each between a `preserve` and a `restore`.
- For each table, we start by using the SSC packages `sdecode` to decode the model–parameter variable `modparam` to a row–label variable `rowlab`, and `bmjcip` to decode the estimates, confidence limits, and *P*–values.
- We then use the SSC package `chardef` to set **column characteristics**, specifying the column labels and alignments.
- In the first table, we then use the SSC package `insingap` to insert a **gap row** at the start of each car–origin by–group.
- And, in the second table, we then use the SSC package `xrewide` (an extension of `reshape wide`) to reshape the table to give the 2 car–origin groups side by side.

Creating a Markdown document from our resultsset

- The do–file `mymddoc1.do` generates a HTML document `mymddoc1.htm` (via a Markdown document `mymddoc1.md`) from our resultsset in `result1.dta`.
- We use the SSC package `tfinsert` to write the text, and then create 2 alternative tables from our resultsset using `listtab`, each between a `preserve` and a `restore`.
- For each table, we start by using the SSC packages `sdecode` to decode the model–parameter variable `modparam` to a row–label variable `rowlab`, and `bmjcip` to decode the estimates, confidence limits, and *P*–values.
- We then use the SSC package `chardef` to set **column characteristics**, specifying the column labels and alignments.
- In the first table, we then use the SSC package `insingap` to insert a **gap row** at the start of each car–origin by–group.
- And, in the second table, we then use the SSC package `xrewide` (an extension of `reshape wide`) to reshape the table to give the 2 car–origin groups side by side.

Creating a Markdown document from our resultsset

- The do–file `mymddoc1.do` generates a HTML document `mymddoc1.htm` (via a Markdown document `mymddoc1.md`) from our resultsset in `result1.dta`.
- We use the SSC package `tfinsert` to write the text, and then create 2 alternative tables from our resultsset using `listtab`, each between a `preserve` and a `restore`.
- For each table, we start by using the SSC packages `sdecode` to decode the model–parameter variable `modparam` to a row–label variable `rowlab`, and `bmjcip` to decode the estimates, confidence limits, and *P*–values.
- We then use the SSC package `chardef` to set **column characteristics**, specifying the column labels and alignments.
- In the first table, we then use the SSC package `insingap` to insert a **gap row** at the start of each car–origin by–group.
- And, in the second table, we then use the SSC package `xrewide` (an extension of `reshape wide`) to reshape the table to give the 2 car–origin groups side by side.

Creating a Markdown document from our resultsset

- The do–file `mymddoc1.do` generates a HTML document `mymddoc1.htm` (via a Markdown document `mymddoc1.md`) from our resultsset in `result1.dta`.
- We use the SSC package `tfinsert` to write the text, and then create 2 alternative tables from our resultsset using `listtab`, each between a `preserve` and a `restore`.
- For each table, we start by using the SSC packages `sdecode` to decode the model–parameter variable `modparam` to a row–label variable `rowlab`, and `bmjcip` to decode the estimates, confidence limits, and *P*–values.
- We then use the SSC package `chardef` to set **column characteristics**, specifying the column labels and alignments.
- In the first table, we then use the SSC package `insingap` to insert a **gap row** at the start of each car–origin by–group.
- And, in the second table, we then use the SSC package `xrewide` (an extension of `reshape wide`) to reshape the table to give the 2 car–origin groups side by side.

Decoding the row label and cell variables using `sdecode` and `bmjcip`

To make the first table, we use `sdecode` to decode the model–parameter variable `modparam` to a row–label string variable `rowlab`, and then use the `sdecode`–dependent package `bmjcip` to decode the estimates, confidence limits, and *P*–values (to HTML). We then `list` the car–model origin group variable `us2` and the variables to be tabulated:

```
. sdecode modparam, gene(rowlab);
. bmjcip estimate min* max* p, esub(htmlsuper);
. list us2 rowlab estimate min* max* p, noobs abbr(32);
```

us2	rowlab	estimate	min95	max95	p
Non-US (N=22)	Weight (US tons)	11.059	(8.130,	13.987)	1.5x10⁻⁷
Non-US (N=22)	Constant	-1.764	(-5.382,	1.853)	.32
US (N=52)	Weight (US tons)	7.885	(6.484,	9.286)	2.2x10⁻¹⁵
US (N=52)	Constant	0.537	(-1.549,	2.622)	.61

Note that the confidence limits now have commas and parentheses, and the *P*–values are already in HTML.

Setting the column–variable characteristics using `chardef`

The SSC package `chardef` is a mass–production extension of `char define`. We use it to set the values of the variable characteristics `varname` (containing the italicized column headings in Markdown) and `halign` (containing column horizontal alignments in Markdown) for the variables `rowlab`, `estimate`, `min95`, `max95`, and `p`, which will eventually appear in the table:

```
. chardef rowlab estimate min* max* p, char(varname)
>   val("Parameter" "Estimate" "(95%" "CI)" P) prefix(*) suffix(*);
. chardef rowlab estimate min* max* p, char(halign)
>   val(":---" "---:" "---:" "---:" ":---");
```

If you understand Markdown, then you might note that the options `prefix(*)` and `suffix(*)` of `chardef` are used to make the column headings italic. And that the preferred horizontal alignments for the variables `rowlab`, `estimate`, `min95`, `max95`, and `p` are left, right, right, right, and left, respectively.

Inserting a single gap row in each by–group using `insingap`

The SSC package `insingap` depends on the SSC package `ingap`, which in turn depends on the SSC package `sdecode`. We use it to insert a gap row at the beginning of each by–group defined by the car–origin group variable `us2`. We then `list` the variables which will eventually be output to the table in the Markdown document:

```
. insingap us2, rowlabel(rowlab) grdecode(us2) prefix(**) suffix(:**)
>   neword(rowseq);
. list rowlab estimate min* max* p, noobs abbr(32) sepby(us2);
```

rowlab	estimate	min95	max95	p
Non-US (N=22):				
Weight (US tons)	11.059	(8.130,	13.987)	1.5x10⁻⁷
Constant	-1.764	(-5.382,	1.853)	.32
US (N=52):				
Weight (US tons)	7.885	(6.484,	9.286)	2.2x10⁻¹⁵
Constant	0.537	(-1.549,	2.622)	.61

If you understand Markdown, then you might note that the `prefix(**)` and `suffix(:**)` options of `insingap` are used to make the gap row labels bold (with colons).

Outputting the table to the Markdown document using `listtab`

We can now output the table defined by the *varlist*
`rowlab estimate min* max* p`
to the Markdown document under construction. The table is also typed (thanks to the `type` option) to the Stata log, where it can be seen as an alien–looking Markdown table, ready for conversion to an even more alien–looking HTML table:

```
. listtab rowlab estimate min* max* p, handle('mdb1') rstyle(markdown)
>   headchar(varname halign) type;
|*Parameter*|*Estimate*|*(95%*|*CI)*|*P*|
|:---|---:|---:|---:|:---|
|**Non-US (N=22):**|||||
|Weight (US tons)|11.059|(8.130,|13.987)|1.5x10<sup>-7</sup>|
|Constant|-1.764|(-5.382,|1.853)|.32|
|**US (N=52):**|||||
|Weight (US tons)|7.885|(6.484,|9.286)|2.2x10<sup>-15</sup>|
|Constant|0.537|(-1.549,|2.622)|.61|
```

If you understand Markdown, then you might note that the table starts with an italic column–label row, continues to a column–alignment row, and then continues with the results rows, including bold gap rows with colons. And, if you *don't* understand Markdown. . .

We can now view the browser–ready HTML document `mymddoc1.htm`

- The second table (the wide version) is created using `xrewide` instead of `insingap`, but is otherwise similar.
- We can now look at the browser–ready document `mymddoc1.htm`, containing both tables.

We can now view the browser–ready HTML document `mymddoc1.htm`

- The second table (the wide version) is created using `xrewide` instead of `insingap`, but is otherwise similar.
- We can now look at the browser–ready document `mymddoc1.htm`, containing both tables.

We can now view the browser–ready HTML document `mymddoc1.htm`

- The second table (the wide version) is created using `xrewide` instead of `insingap`, but is otherwise similar.
- We can now look at the browser–ready document `mymddoc1.htm`, containing both tables.

Browser–ready HTML documents *versus* printer–ready `.docx` documents

- Our superiors frequently prefer printer–ready documents to browser–ready documents.
- A printer–ready document must have a page size (such as A4), and usually a page header and/or a page footer.
- The header may contain corporate logos for organizations participating in the project.
- And the header and/or the footer may contain a page number and a page count. (These are useful if our superiors print the document and scatter the pages over the floor.)
- In Stata, we can create printer–ready `.docx` documents, using `putdocx`.
- And to create `.docx` tables in such documents from resultssets, we use the SSC package `docxtab`.

Browser–ready HTML documents *versus* printer–ready `.docx` documents

- Our superiors frequently prefer printer–ready documents to browser–ready documents.
- A printer–ready document must have a page size (such as A4), and usually a page header and/or a page footer.
- The header may contain corporate logos for organizations participating in the project.
- And the header and/or the footer may contain a page number and a page count. (These are useful if our superiors print the document and scatter the pages over the floor.)
- In Stata, we can create printer–ready `.docx` documents, using `putdocx`.
- And to create `.docx` tables in such documents from resultssets, we use the SSC package `docxtab`.

Browser–ready HTML documents *versus* printer–ready `.docx` documents

- Our superiors frequently prefer printer–ready documents to browser–ready documents.
- A printer–ready document must have a page size (such as A4), and usually a page header and/or a page footer.
- The header may contain corporate logos for organizations participating in the project.
- And the header and/or the footer may contain a page number and a page count. (These are useful if our superiors print the document and scatter the pages over the floor.)
- In Stata, we can create printer–ready `.docx` documents, using `putdocx`.
- And to create `.docx` tables in such documents from resultssets, we use the SSC package `docxtab`.

Browser–ready HTML documents *versus* printer–ready `.docx` documents

- Our superiors frequently prefer printer–ready documents to browser–ready documents.
- A printer–ready document must have a page size (such as A4), and usually a page header and/or a page footer.
- The header may contain corporate logos for organizations participating in the project.
- And the header and/or the footer may contain a page number and a page count. (These are useful if our superiors print the document and scatter the pages over the floor.)
- In Stata, we can create printer–ready `.docx` documents, using `putdocx`.
- And to create `.docx` tables in such documents from resultssets, we use the SSC package `docxtab`.

Browser–ready HTML documents *versus* printer–ready `.docx` documents

- Our superiors frequently prefer printer–ready documents to browser–ready documents.
- A printer–ready document must have a page size (such as A4), and usually a page header and/or a page footer.
- The header may contain corporate logos for organizations participating in the project.
- And the header and/or the footer may contain a page number and a page count. (These are useful if our superiors print the document and scatter the pages over the floor.)
- In Stata, we can create printer–ready `.docx` documents, using `putdocx`.
- And to create `.docx` tables in such documents from resultssets, we use the SSC package `docxtab`.

Browser–ready HTML documents *versus* printer–ready `.docx` documents

- Our superiors frequently prefer printer–ready documents to browser–ready documents.
- A printer–ready document must have a page size (such as A4), and usually a page header and/or a page footer.
- The header may contain corporate logos for organizations participating in the project.
- And the header and/or the footer may contain a page number and a page count. (These are useful if our superiors print the document and scatter the pages over the floor.)
- In Stata, we can create printer–ready `.docx` documents, using `putdocx`.
- And to create `.docx` tables in such documents from resultssets, we use the SSC package `docxtab`.

Browser–ready HTML documents *versus* printer–ready `.docx` documents

- Our superiors frequently prefer printer–ready documents to browser–ready documents.
- A printer–ready document must have a page size (such as A4), and usually a page header and/or a page footer.
- The header may contain corporate logos for organizations participating in the project.
- And the header and/or the footer may contain a page number and a page count. (These are useful if our superiors print the document and scatter the pages over the floor.)
- In Stata, we can create printer–ready `.docx` documents, using `putdocx`.
- And to create `.docx` tables in such documents from resultssets, we use the SSC package `docxtab`.

Creating a `.docx` document from our resultsset

- The do–file `mydocxdoc1.do` generates a `.docx` document `mydocxdoc1.docx` from our resultsset in `result1.dta`.
- This time, we use `putdocx` to write the text, and create the 2 alternative tables from our resultsset using `docxtab`, each between a `preserve` and a `restore`.
- Otherwise, we use similar methods to the Markdown example, using `sdecode`, `bmjcip`, `chardef`, `insingap`, and `xrewide`.
- These methods form a variant of the DCRIL path, where `docxtab` does the listing.
- We can now view the `.docx` document created.

Creating a `.docx` document from our resultsset

- The do–file `mydocxdoc1.do` generates a `.docx` document `mydocxdoc1.docx` from our resultsset in `result1.dta`.
- This time, we use `putdocx` to write the text, and create the 2 alternative tables from our resultsset using `docxtab`, each between a `preserve` and a `restore`.
- Otherwise, we use similar methods to the Markdown example, using `sdecode`, `bmjcip`, `chardef`, `insingap`, and `xrewide`.
- These methods form a variant of the DCRIL path, where `docxtab` does the listing.
- We can now view the `.docx` document created.

Creating a `.docx` document from our resultsset

- The do–file `mydocxdoc1.do` generates a `.docx` document `mydocxdoc1.docx` from our resultsset in `result1.dta`.
- This time, we use `putdocx` to write the text, and create the 2 alternative tables from our resultsset using `docxtab`, each between a `preserve` and a `restore`.
- Otherwise, we use similar methods to the Markdown example, using `sdecode`, `bmjcip`, `chardef`, `insingap`, and `xrewide`.
- These methods form a variant of the DCRIL path, where `docxtab` does the listing.
- We can now view the `.docx` document created.

Creating a `.docx` document from our resultsset

- The do–file `mydocxdoc1.do` generates a `.docx` document `mydocxdoc1.docx` from our resultsset in `result1.dta`.
- This time, we use `putdocx` to write the text, and create the 2 alternative tables from our resultsset using `docxtab`, each between a `preserve` and a `restore`.
- Otherwise, we use similar methods to the Markdown example, using `sdecode`, `bmjcip`, `chardef`, `insingap`, and `xrewide`.
- These methods form a variant of the DCRIL path, where `docxtab` does the listing.
- We can now view the `.docx` document created.

Creating a `.docx` document from our resultsset

- The do–file `mydocxdoc1.do` generates a `.docx` document `mydocxdoc1.docx` from our resultsset in `result1.dta`.
- This time, we use `putdocx` to write the text, and create the 2 alternative tables from our resultsset using `docxtab`, each between a `preserve` and a `restore`.
- Otherwise, we use similar methods to the Markdown example, using `sdecode`, `bmjcip`, `chardef`, `insingap`, and `xrewide`.
- These methods form a variant of the DCRIL path, where `docxtab` does the listing.
- We can now view the `.docx` document created.

Creating a `.docx` document from our resultsset

- The do–file `mydocxdoc1.do` generates a `.docx` document `mydocxdoc1.docx` from our resultsset in `result1.dta`.
- This time, we use `putdocx` to write the text, and create the 2 alternative tables from our resultsset using `docxtab`, each between a `preserve` and a `restore`.
- Otherwise, we use similar methods to the Markdown example, using `sdecode`, `bmjcip`, `chardef`, `insingap`, and `xrewide`.
- These methods form a variant of the DCRIL path, where `docxtab` does the listing.
- We can now view the `.docx` document created.

References

[1] Newson, R. B. Resultssets in resultsframes in Stata 16–plus. Presented at the 2022 London Stata Conference, 8–9 September, 2022. Downloadable from the conference website at *https://econpapers.repec.org/paper/boclsug22/01.htm*

[2] Date, C. J. *An Introduction to Database Systems. 8th Edition.* Pearson Education, Inc.; 2003.

[3] Newson, R. B. 2012. From resultssets to resultstables in Stata. *The Stata Journal* **12(2)**: 191—213. Downloadable from *https://journals.sagepub.com/doi/pdf/10.1177/1536867X1201200203*

[4] Newson, R. B. Creating factor variables in resultssets and other datasets. Presented at the 19th UK Stata User Meeting, 12–13 September, 2013. . Downloadable from the conference website at *http://ideas.repec.org/p/boc/usug13/01.html*

[5] Newson, R. B. 2017. Stata Tip 127: Use `capture noisily` groups. *The Stata Journal* **17(2)**: 511–514. Downloadable from *https://journals.sagepub.com/doi/pdf/10.1177/1536867X1701700215*

The presentation, and the example do–files, can be downloaded from the conference website. The packages can be downloaded from SSC.