

MDR Functional User's Handbook

Version 2.4

Prepared for: Office of the Assistant Secretary of Defense (Health Affairs) /
TRICARE Management Activity

Updated: 04/30/2010

Table of Contents

PURPOSE OF THIS DOCUMENT	4
OVERVIEW OF THE MHS DATA REPOSITORY	4
Background (Definition of the MDR)	4
MDR Environment	4
Overview of Data Types	4
Purchased Care	4
Direct Care (MTF)	5
DEERS	6
PDTs	6
Directory Structure of the MDR	6
Overview of All MDR Directories	6
Public Directories (MDR PUB)	7
MDR Resources	8
Interface Control Documents, MDR Functional Specifications, M2 Functional Specifications, and MDR Data Dictionary	8
BASIC UNIX	9
What is Unix?	9
Common Unix Commands	9
PICO Editor	12
HOW TO WRITE/MODIFY AND RUN SAS PROGRAMMING AND THE MDR	12
Programming 101	12
SAS Logs and SAS Lists	13
Process of Developing Programming Code	13
SAS Options	13
Types of Data Files and File Declaration	14
Review of SAS Options and File Declaration	15
Documenting Your Code	16
SAS Inputs	16
Inputting Members of SAS Data Sets	16
SAS Data Libraries and Members	16
Text Files	17
Input Statements (Delimited Data)	19
Efficiency and Input Statements	19
Conditional Input Statements	20
Conditional Inputs for Text Files	20
Programming Technique: After Data is Input, What Next?	21
Functions	21
Math	21
Strings	21
If Then Else	22
Comparison Operators	23
Formats and the Put Statement	23
Date Functions and Formats	24
Procedures	25
Contents	26
Print	28
Sort	29
Freq	30
Summary	31
Format	33
Combining Data Sets	34

Appending Datasets.....	34
Merging Datasets	35
Arrays.....	37
Macros	40
Available SAS Macros.....	40
User Defined Macros	41
Output Data Sets	44
Writing SAS Datasets.....	44
Quality Review	47
Documenting Programs	48
Testing Code on Subsets of Data	48
Confirmation of Inputs and Outputs	48
Logic Checks.....	48
Track Data Flow	49
Review Logs	49
Produce Summary Statistics and Frequencies	49
Comparative Sources	49
MDR PROGRAMS	50
Programming Examples	50
Appendix A: Comparison of MDR Data files and M2 Data Files	50
Appendix B: Access Permissions	52
chmod 764 tedn.sas - allows owner read/write/execute, allows group read/write only, allows other read only	52
Appendix C: Sample Programs	53

List of Tables

Table	Title	Page
1	The Major CHCS Extracts	5
2	MDR Directories	6
3	MDR Pub Directory	7
4	MDR Ref Directory	8
5	MDR Resources	8
6	Unix Commands	9
7	PICO Editor Cheat Sheet	12
8	Comparison Operators	23
9	Summarized SAS Commands	25
10	Most Common Procedures Used in MDR	25
11	Including Records Using 'If' Statement	36

PURPOSE OF THIS DOCUMENT

This document is prepared for new functional users of the MDR. If you've just received your password and have all the software loaded and ready to go, this document is for you! Welcome! The body of the document describes the basic framework of the MDR, how to move around in it (you can't just point / click / drag / drop!), how to manage files and directories, and how to write and run basic SAS code using MDR files. Examples of programming code are included throughout this document.

OVERVIEW OF THE MHS DATA REPOSITORY

Background (Definition of the MDR)

The MHS Data Repository (MDR) is a data warehouse containing the most complete collection of data about healthcare provided to beneficiaries of the MHS. The MDR receives data from a wide variety of sources, throughout the enterprise, and processes these sources according to a set of published business rules. Information in the MDR is made available to a set of 'super users' via the SAS programming language, and also through files that are prepared to be used/displayed in other systems.

MDR Environment

For more information about the technical aspects of the MDR, consult the slides, any documentation provided by DHSS when you receive your account, and the MDR User's Guide. For problems with connection or access, contact the Help Desk at 1-800-600-9332.

Overview of Data Types

The MDR has been operational since 1999. There have been many changes and enhancements to the system since its initial implementation. Most materials focus on the most recent versions of data files, but sometimes data are different over time. When using older data, it is important to analyze it carefully (feel free to contact Kennell) to discuss important differences that may affect your results.

Purchased Care

There are many files in the MDR related to purchased care. Claims files and provider files are both generally available. The claims files include extracts of claims that were received and paid for particular types of services / beneficiaries, while the provider files provide detailed information about those providing the care to MHS eligibles.

TRICARE Medical Claims are sent from the Managed Care Support Contractors on a daily basis. These claims are sent into the PEPR (Aurora system) daily. TMA staff batch up claims records and related provider information once per month and send to the MDR. The types of files sent include:

- Institutional Claims: Inpatient Institutional (Hospital, Rehab, Skilled Nursing Facility, etc) Claims and Home Health Claims
- Non-Institutional Claims: All other TRICARE medical (and pharmacy claims)
- Provider Records: Providers that can bill TRICARE.

Designated Provider Claims and Provider Files are sent from the Designated Provider Data Processing Contractor to the MDR once per month.

MMSO Active Duty Purchased Care Dental Files are sent from the Military Medical Support Office once per month. These have since been replaced by the Active Duty Dental Plan (ADDP) data, also received once per month.

Direct Care (MTF)

Most of the data files about care provided by direct care facilities come from the local Composite Health Care System (CHCS) servers in the form of routine, standardized extracts. These extracts include details about care provided by organizations using CHCS; mostly military treatment facilities (to limit to just MTF, use the MTF Branch of Service data element).

The major CHCS extracts in the MDR include:

Table 1: The Major CHCS Extracts

Extract	Nickname	Record Def	Sent	Processed
Standard Inpatient Data Record	SIDR	Inpatient Hosp Record	Monthly	Monthly
Appointment		Appointment	Weekly	Monthly
Referral		Referrals	Weekly	Weekly
Standard Ambulatory Data Record	SADR	Professional Outpatient + Rounds	Daily	Weekly
Comprehensive Professional Record	CAPER	Enhanced SADR	Daily	Weekly (Semi)
Ancillary		Outpatient Lab/Rad/Rx ¹	Monthly	Monthly
Worldwide Workload Report	WWR	Summary Workload	Monthly	Monthly

For each of these files, the MDR contains enhanced data files that include popular data elements such as RVUs, RWPs and standardized demographics from DEERS, where possible.

Some of these data files are processed further to create additional files. The case management episode file, and the CHCS address file (restricted) are two examples.

The Expense Assignment System (EAS) is a local management accounting system (term used somewhat loosely) used by MTFs to keep track of expenses and obligations and full-time equivalent staff information. EAS does not contain person or event level data. EAS sends extracts to the MDR allowing for reporting by MTF, year, month and accounting code (MEPRS code).

There are two EAS extracts, in addition to a few reference tables. For most purposes, the main MEPRS extract will suffice. This extract includes workload, costs and full-time equivalent data. An additional personnel extract was added to the MDR recently. This extract contains the same full-time equivalent data as the main MEPRS file, but there is additional detail available.

There is no Direct Care provider table in the MDR, though provider information is collected centrally from CHCS and could be made available rather quickly.

¹ The ancillary rx table sent from CHCS is not stored in the MDR as a separate file; instead it is merged into the MDR Pharmacy file and some of its contents are added to each direct care pharmacy record in PDTS.

DEERS

DEERS is a component of DoD responsible for managing information about benefits received through an association with DoD. One such benefit is health care. DEERS provides 2 files to the MDR each month.

VM6: The DEERS VM6 file is a beneficiary-level file sent from DEERS to the MDR each month. The feed from DEERS contains one record for each beneficiary relationship in DEERS. Many of the records are removed during MDR processing though, such that what remains after processing includes at least one record for each beneficiary who:

- has any type of eligibility on the 1st of the reported month
- is in the guard or reserve or sponsored by a guard or reserve member
- is a member of a family where any beneficiary has eligibility

To limit the DEERS VM6 Records to eligible beneficiaries, use the MHS eligibility indicator. To limit to only one record per person (the record with the highest benefit level), use the primary record flag.

The VM6 file is used to prepare many outputs, including a TRICARE longitudinal eligibility file, a death file, and a special enrollment file

Reservist File: The Reservist file is a comprehensive file with all guard/reserve activations since 9/11/2001. The file includes begin and end dates as well as a status code that indicates early eligibility, deployment period or transitional assistance.

All files from DEERS are processed monthly.

PDTS

The Pharmacy Data Transaction Service provides drug utilization review for the MHS. All outpatient prescriptions (except those provided by civilian pharmacies overseas) are included in the PDTS database. Once a week, PDTS sends a data file to the MDR, including all new and reversed (never picked up) prescriptions since the previous week. PDTS includes scripts dispensed at MTFs, MCSCs and TMOP.

Directory Structure of the MDR

When users are granted a password to the MDR, and all the proper software and security requirements have been met, a directory will be established for the new user. The user will also be assigned access privileges to other needed directories. Default access includes a user home directory, an organizational directory (i.e. company or project) and access to the 'public' and 'reference' files of the MDR. To access additional directories, special justification must be provided to TMA/BEA (jamie.lindly@tma.osd.mil) for routing and approval.

[Overview of All MDR Directories](#)

Table 2: MDR Directories

Directory Name	Access	Content
PUB	Default	MDR Analytical Data Sets
REF	Default	Reference Files

Directory Name	Access	Content
RESTRICTED	Special Just	Sensitive MDR Analytical Data Sets
PROD	MDR Project Files; not generally accessible	
LOG		
INTERIM		
RAW		
ARS	Special Just	Data feeds for M2
APUB	Special Just	Archives of public files
AREF	Special Just	Archives of reference files

This document will focus on the default access directories.

HOME DIRECTORY:

The storage provided in this directory is small. DHSS has recommended that this directory generally not be used.

ORGANIZATIONAL OR PROJECT DIRECTORY:

This directory is where users will store all of their personal files; including programs, interim and final processing products and other files. Most organizational directories also have a structure to them, specific to each organization.

Public Directories (MDR PUB)

This directory contains the MDR processed analytical data files. MDR users can write programs using the files in MDR PUB by specifying the location of the file they want to use (directory name) and the file name needed in their programs. This will be discussed in detail later. BEA provides the MDR User's Guide to new users that includes a complete listing of the directories in MDR PUB. The most commonly used directories and the file names within them are:

Table 3: MDR Pub Directory

Content	MDR Directory Path and File Name
DEERS Beneficiary Level	/mdr/pub/deers/detail/vm6ben/fy<fy>/fm<fm>.txt
DEERS Enrollment	/mdr/pub/deers/enr/vm6enr/fy<fy>/fm<fm>.sas7bdat
Case Management	/mdr/pub/casemgmt/cm.sas7bdat
MTF Ancillary Lab & Rad	/mdr/pub/ancillary/fy<fy>/ancillary.sas7bdat
MTF Inpatient	/mdr/pub/sidr/fy<fy>/sidr.fy<fy>/fy<fy>.sas7bdat
MTF Professional	/mdr/pub/sadr/fy<fy>.sas7bdat
TED Institutional	/mdr/pub/tedi/fy<fy>/header.sas7bdat
TED Non-Inst DHP	/mdr/pub/tedni/fy<fy>/champus.sas7bdat
TED Non-Inst MERHCF	/mdr/pub/tedni/fy<fy>/tdefic.sas7bdat
Pharmacy	/mdr/pub/pdts/detail/fy<fy>/pdts.detail.fy<fy>.txt.Z
Referral	/mdr/pub/referral/referral.sas7bdat
MEPRS	/mdr/pub/eas4/fy<fy>/eas4.fy<fy>/fy<fy>.sas7bdat

Reference Directories (MDR REF):

This directory contains two different types of data files. There are reference tables that are used in MDR processing and also executable SAS format statements which can be included in user programs to easily append reference data. The complete list of reference tables is

also described in the guide that BEA provides new users. Selected files in the MDR REF directory are:

Table 4: MDR Ref Directory

Content	MDR Directory Path and File Name
APG Weight Table	/mdr/ref/apgref.txt
CPT Weight Table	/mdr/ref/cptref.cy<cy>.txt
DMISID Reference Table	/mdr/ref/dmisid.index.fy<fy>.txt
DRG Weight Tables	/mdr/ref/drgref.fy<fy>.txt
ICD-9 Diagnosis Descriptions	/mdr/ref/icd9dxref.fy<fy>.txt
ICD-9 Procedure Descriptions	/mdr/ref/icd9proref.fy<fy>.txt
Catchment Area Directory	/mdr/ref/cad.omni/a<cycm>.sas7bdat
MEPRS 3 Codes Descriptions	/mdr/ref/eas4.mepr3.fy<fy>/fy<fy>.sas7bdat
MEPRS 4 Codes Descriptions	/mdr/ref/eas4.mepr4.fy<fy>/fy<fy>.sas7bdat
Purchased Care Provider Table	/mdr/pub/tedpr/tedpr.sas7bdat

The executable SAS code files will be discussed later in this document, as the application of them is described.

For M2 users, Appendix A contains a table which lists the files in M2, the corresponding source files in the MDR, and additional information about how to filter data in the MDR files to match the criteria used for M2, if needed.

MDR Resources

There are many resources available to help users to understand the data in the MDR. There is no training course designed to instruct in detail on the content and functional application of MDR data files. However, the WISDOM course contains extremely useful functional information about the content of the M2 (data from MDR). While the data files in MDR and M2 are not always exactly the same, the WISDOM course would provide a good foundation for understanding much of the important data in the MDR. Data files are not always exactly the same, though. Refer to Appendix A for information about how MDR and M2 files compare.

At http://www.tricare.mil/ocfo/bea/functional_specs.cfm, many electronic resources can be found. This website is prepared for the functional proponent of the MDR, TMA/ Business and Economic Analysis (BEA).

[Interface Control Documents, MDR Functional Specifications, M2 Functional Specifications, and MDR Data Dictionary](#)

Table 5: MDR Resources

Type of File	Purpose
Interface Control Documents	Describes flow of data from sources to MDR
MDR Functional Specifications	Describes processing of data to prepare files in mdr/pub
M2 Functional Specifications	Describes M2 extract and linking
MDR Data Dictionary	Describes content of each data element in each file in MDR

The MDR Functional Specifications and/or the MDR Data Dictionary are important resources for programmers and will be used throughout this document.

BASIC UNIX

What is Unix?

Since there is no point and click access to the MDR, users must learn to use UNIX commands to work on the MDR. UNIX is extremely robust and there are literally thousands of UNIX books that can be purchased. Note that UNIX is case sensitive. Do not capitalize or the system will not understand what you are asking it to do. The table below summarizes select UNIX commands and provides information on how they are used by MDR users.

Common Unix Commands

Table 6: Unix Commands

Action Required	Statement	Notes
Print working directory	pwd	Tells you the location of your current directory
Change access permissions for a file or directory you have created.	chmod <i>* * *</i> dirname or filename	See Appendix B
List contents of a directory	ls	List directories & files (no details)
	ls -l	List with details (permission, owner, size, date)
	ls -lrt	List with details sorting by ascending date
Change directories	cd cd ~	Go to home directory
	cd /	Go to root directory
	cd .. cd ../../	Go back 1 directory Go back 2 directories
	cd dirname	Go to directory "dirname"
	cd /mdr/pub/tedni	Go to directory MDR pub TEDni
Copy a file	cp oldfile newfile cp -i oldfile newfile	Defaults to using the same filename, if you do not specify a new name change. Can use a wildcard. Option -i will prompt user to overwrite newfile if one already exists.
Move or rename a file	mv oldfile newfile mv -i oldfile newfile	Defaults to using the same filename, if you do not specify a new name change. Can use a wildcard. Option -i will prompt user to overwrite newfile if one already exists.
Make a new directory	mkdir newname	Creates a new directory. Access permissions default to full access for all users in the

Action Required	Statement	Notes
		organization as the 'originator' of the directory and no access for others.
Delete a file	rm filename	Unix won't ask for confirmation so BE CAREFUL!!
Delete a directory	rmdir dirname	Must clear all contents within this directory first.
Lists running jobs and associated statistics	llq	Used to track whether a program is still running
Look at the contents of an uncompressed file (uncompressed)	pico filename more filename (all of it) head filename (1 st ten rows displayed onscreen) head filename > tempfile.txt (1 st 10 rows written out to a file) tail filename (last ten rows displayed onscreen) tail filename > tempfile.txt (last 10 rows written out to a file)	Only works when files are uncompressed. Does not work on SAS. Also, large files may exceed available buffer.
See amount of available disk space in a directory	df -k directory_name	
Executes a SAS program	sas program_name sasbig program_name	The "Load Leveler" requires the program name to end in ".sas". Choose sas or sasbig depending on temporary workspace requirement.
Stop a running program	sasstop job_id	Stops a program from running. Use 'llq' to get job_id.

When users first access the MDR, a login screen will pop up. After entering the user ID, reading the warning banner and entering the password, the user will see only a UNIX prompt. Some users will see a directory name (representing the home directory *userid*) followed by a UNIX prompt. A UNIX prompt is a \$. UNIX commands are entered after the \$.

The first UNIX command a new MDR user will generally do is "cd". The CD command allows the user to move between directories. Since users are granted access to a default organizational or project directory, this directory is usually the first place a user moves to when in the MDR. To move to this directory, use the cd command followed by a space, and then the directory path that you wish to move to. For example,

```
cd /hpae2
```

The directory name being accessed here is actually `hpae2`, but notice that the statement is preceded by a `/`. The `/` is always required when moving from one "root" directory to another. Root directories are the most structured directories within the MDR². Your organizational directory is likely a root directory, as is the case in the example above. `'hpae2'` is an organizational directory. Most organizational directories also have subdirectories. In the case of the `/hpae2` directory, there are subdirectories that represent the various companies working for the organization. These subdirectories are usually established by the system administrators to ensure separation of data as needed.

Once inside a root directory, the additional `/` is not needed to move from one subdirectory to another. For example, once inside `cd/hpae2`, `cd kennell` would move the user into the `/hpae2/kennell` directory. To move backwards, use `"cd.."`. (A `cd` with two periods after it).

Another extremely important UNIX command is `"ls"`; which lists the contents of the directory you are in. Just like your PC, this listing will include directories and files. Also, like your PC, if you'd like to see more details about files you can specify parameters. Most users will use the `"-lrt"` option to show file size and date information in addition to names.

```
ls -lrt
```

After entering `'ls -lrt'`, the user will see something like:

```
drwxr-s--- 2 wfunk shken      256 Sep 04 07:13 pharmacy
-rw-r----- 1 wfunk shken     4191 Jul 31 04:50 tedn.sas
```

The first character (`d` or `-`) tells whether the item listed is a directory or a file. The letter `"d"` in the `"drwxr-s---"` sequence indicates that `'pharmacy'` is a directory. The `'-'` in the row below indicates that `tedn.sas` is a file. The rest of that sequence of (`drwxr-s---` and `-rw-----`) indicates access permissions to the listed file. See Appendix B for more information on what these mean and how to change them.

Using the `"cd"`, `"cd.."` and `"ls -lrt"` commands is akin to the screens you see when clicking around in Microsoft Explorer on your PC – it is just not user friendly!

The command `"mkdir"` and `"rmdir"` make and remove (delete) directories in the directory the command is issued from. For example, if a user was located in the directory `'/hpae2/kennell'`, then after issuing the command `'mkdir wendy'`, the `'ls'` command would show a new directory called `'wendy'`.

To copy or move files, the commands `"cp"` and `"mv"` are used, respectively. The command `'cp wendy.sas /wendy/ted.sas'` would copy the file `'wendy.sas'` in the current directory, to a subdirectory called `Wendy`. The file would no longer be named `wendy.sas`, but instead `ted.sas`.

To delete a file, use the command `'rm filename'`.

`More`, `head`, and `tail` followed by a file name open up the specified file for viewing. The `'more'` command opens the entire file. The `'head'` and `'tail'` commands show the first and last ten rows of a file respectively.

² `"mdr"` is the root directory that contains `/mdr/pub` and `/mdr/raw` described earlier in this document.

To run a SAS program, use the command "sas". The command 'sas sidr.sas' would run a program called sidr.sas from the current directory. Use 'llq' to monitor the status of your jobs. See the MDR User's Guide for the command line used to submit SAS jobs when implementing the suggested "Keeping Logs" macro.

PICO Editor

There are many different methods that can be used to create SAS computer programs. Programs can be written on your PC using word processing software and then uploaded, you can copy code (using 'cp') from somewhere else. You can also write code in PC SAS to take advantage of its easy interface. Finally, and often most practical, you can use editors available on the MDR: either VI or PICO. It is often advantageous to use a combination of methods.

The PICO editor is easy to use and often indispensable when debugging a program. To create a file, simply type 'pico filename'. SAS programs should end in the '.sas' extension. Once in the PICO editor, simply type your code. The [control] key is used as noted in the table below to move around within the editor. A "cheat sheet" of the functionality associated with control keys is located on the bottom of the screen when in the PICO editor.

Table 7: PICO Editor Cheat Sheet

Action Required	Statement	Notes
Create and edit a file (.sas, .log, .lst)	pico filename	Invokes the pico editor. There are "control" keys on the bottom of the screen that guide you.
Copy and paste	Highlight what you want to copy you're your mouse, move cursor to where you want it copied with arrow keys, and "right click"	The mouse will not move you to where you want to be, only the arrow keys or page up/down.
Get in/out of PICO editor	[Ctrl] X	Need to hold down the [Ctrl] key and press the desired key
Search/Find text in the file	[Ctrl] W	
Go to end of the line	[Ctrl] E	
Delete a line	[Ctrl] K	
Undo delete a line	[Ctrl] U	
Cut and paste	[Ctrl] ^, use arrow keys to highlight, [Ctrl] K to "cut", use arrow keys to move to where you want to paste, [Ctrl] U to paste	Ctrl ^ marks the beginning of what you want to copy,

HOW TO WRITE/MODIFY AND RUN SAS PROGRAMMING AND THE MDR

Programming 101

Writing computer programs is usually an iterative process. It is the rare programmer who can simply sit down and write out a perfect computer program on the first try! Most programmers work iteratively, testing new lines of code on small bits of data piece by piece. This is very good practice, especially for those new to programming.

SAS is an extremely powerful language and can be used for very simple tasks, along with complex analytical tasks. The SAS language itself works rather simply. Programmers write a series of instructions to the computer using the rules of SAS. Fundamental to the SAS language is the creation of working data sets to use for further processing. These data sets are created in what is called 'the data step'. SAS programs usually involve the creation of one or many working data sets (that is, data steps). Once a data step is done, the SAS program can create variables, combined data sets, perform SAS procedures, use SAS functions, etc. The possibilities are endless.

SAS Logs and SAS Lists

The computer instructions (programs) are submitted to the computer (sas progname.sas); the computer runs the program and returns back a log of how the program behaved; called the SAS Log. The computer will also create print outs and files that were requested in the program submitted to the computer. The user reviews the logs and other files using commands like 'pico' and 'more'. The products that SAS prepares like logs and list files should ALL be carefully reviewed to ensure accuracy of results. This is so fundamentally important it cannot be stressed enough. It often amazes even the best of programmers how easily unexpected problems can crop up!

Process of Developing Programming Code

Most good programmers will develop code by testing programs using small numbers of observations and then hand-checking / visually inspecting in a step by step manner. As each step works successfully, new steps are added and tested. Once the final program is drafted, the number of observations used to test is increased. Depending on the file size and expected length of run time, some users will simply run against the whole database once small numbers of observations are tested successfully. Others use interim runs with random samples to avoid lengthy delays should something unexpected occur. There is no right answer on how to proceed under all circumstances.

SAS Options

Among the first lines of code in most users programs are SAS Options. These are global settings that the user wants applied when the computer program runs. The most commonly used SAS options are used in the SAS statement below:

options nocenter obs=100 fullstimer ps=80 mprint;

There are many other SAS options. See SAS online user help for more information. SAS options can be used to control file sizes, output, memory, how errors are handled, and many other things. The 'options' statement can contain one or as many parameters as the user needs. The 'options' statement must be concluded with a semi-colon.

A description of some commonly used system options are:

- *nocenter*: Left justifies your output on the page. This option is almost always used. Without it, printed output is generally unreadable on the screen.
- *obs=n*: Limits the number of observations in each data step in the program. This option is usually used when testing code (especially important when working with extremely large datasets).

- *errors=n*: Controls the maximum number of observations for which complete error messages are printed. Need to be cautious because input errors can often result in printing of protected health information in SAS logs.
- *fullstimer*: Specifies whether all the performance statistics of your computer system that are available to the SAS System are written to the SAS Log. This is very helpful when testing various methods of processing to determine the most efficient way to proceed. Example of printed output in the SAS Log, if the fullstimer option is used:

NOTE: DATA statement used:

```

Real Time          5:35.25
User CPU Time     1:04.08
System CPU Time   1:27.91
Memory            254k
Page Faults       136296
Page Reclaims    246593
Page Swaps        0
Voluntary Context Switches 7
Involuntary Context Switches 10708

```

- *pagesize=* (or *ps=*): Controls the maximum number of lines per page of output. Minimum value is 15; maximum value is 32767. This is very commonly used if printed output is desired.
- *mprint*: Specifies whether SAS statements generated by macro execution are displayed. Macros are a specific type of SAS utility, which will be addressed (lightly) later in this document.

Types of Data Files and File Declaration

SAS can use many different types of files. A fundamental concept in programming is the declaration of files that the program will use. This step involves naming permanent input and output files that will be used. The basic concept is that in the SAS file declaration, an alias is assigned to each permanent input and output file that will be used in the program. This step creates a shortcut way to reference the file names throughout the program.

The exact language used to declare files depends on whether the file is stored in SAS format, or in text format.

Language for Declaration of SAS Datasets: SAS Datasets are stored in a data structure known as a SAS Library. Files within the library all called "members". When an "ls" command is done in the MDR SIDR area, the file name:

```

/mdr/pub/sidr/fy<fy>/fy<fy>.sas7bdat; e.g.
/mdr/pub/sidr/fy08/fy08.sas7bdat

```

In the file name above, the member is the fy08.sas7bdat. The 'sas7bdat' extension is the easiest way to identify a SAS file. Everything up to the member name is used in the SAS statement 'libname'. The following statement declares a SAS file to be used in the programming code that will follow.

```

libname insidr '/mdr/pub/sidr/fy08';

```

Note that the member name is not included in the 'libname' statement. When this dataset is needed in the programming code that follows, the name 'insidr' will be used to reference it. The term library reference refers to the label, in this case, 'insidr', used as an alias. The libref label must begin with a letter or underscore and must not contain any special characters (only letters, numbers and the underscore).

Language for Declaration of Text Files: Text files do not have a sas7bdat extension. Text files are either stored in compressed format, or not. Compressed files can be recognized by a '.z' as a file extension. Some MDR text files are stored in text format. Some are compressed, some are not. A 'filename' statement is used for declaring both types of text files; but the format of the statement is slightly different.

The following statements are examples of the use of the filename statement for declaring text files:

Uncompressed file:

```
filename cptdata "/mdr/ref/cptref.cy08.txt";
```

The file reference (as opposed to library reference) is 'cptdata'. This label will be used throughout the program to reference this dataset.

For a compressed file, the 'filename' statement is used in combination with the 'uncompress' command in UNIX:

```
filename pdts07 pipe "uncompress -c /mdr/pub/pdts/detail/fy07/pdts.detail.fy07.txt.Z";
```

The word 'pipe' indicates to the SAS compiler that the statements to follow are actually UNIX commands. The 'uncompress' statement uncompresses the file as it is being read in. This process of uncompression is extremely lengthy and it is hoped that some day all compressed files will be removed from the MDR. In the meantime, this technique must be used.

Review of SAS Options and File Declaration

The following is an example of the first few lines of SAS code for the MDR. This code was prepared with techniques that have been covered thus far in this handbook.

```
options nocenter obs=100;
```

```
libname intedi08 '/mdr/pub/tedi/fy08';  
filename inpdts07 pipe "uncompress -c /mdr/pub/pdts/detail/fy07/pdts.detail.fy07.txt.Z";  
filename outdrg "/hpae2/kennell/linda/top20drgs.txt";
```

The 'options' statement in this code ensures that printed output from the program is left-justified when printed or displayed ('nocenter'), and that all datasets used in the program can contain only up to 100 observations ('obs=100').

There are three permanent files used by this program. Two of the files are input files. The 'libname' statement indicates a SAS dataset (in this case, the MDR Institutional TEDs), while the 'filename' statements indicate text files. The first represents an input file that is a SAS dataset; the second represents an input file that is in compressed text format; the last represents an output file to be created by the program.

It is not necessary for the program to use all files that are declared in the 'libname' and/or 'filename' statements in the program.

Documenting Your Code

It is good programming technique to document sections or especially complex lines of code in SAS. This will help others to understand the code. Strings of data between a "*" and a ";" will be ignored by SAS. Also, anything between a "/*" and a "*/" will be ignored.

```
/* This section reads in xxxxxxxx */  
*This is an example of a comment in SAS;
```

You will find comment statements in the programs in the appendix to this document. It is best to get in the practice of using them yourself.

SAS Inputs

SAS Input statements differ, depending on whether the dataset being used is a SAS dataset or a text file. However, in both cases the 'data' statement is used. The statement 'data a;' will create a dataset, stored in temporary member while the program is running, called 'a'.

Inputting Members of SAS Data Sets

The SAS Data Step is the method used by SAS to read data into temporary memory; called 'workspace'. The format to use for the SAS data step is:

data dsname

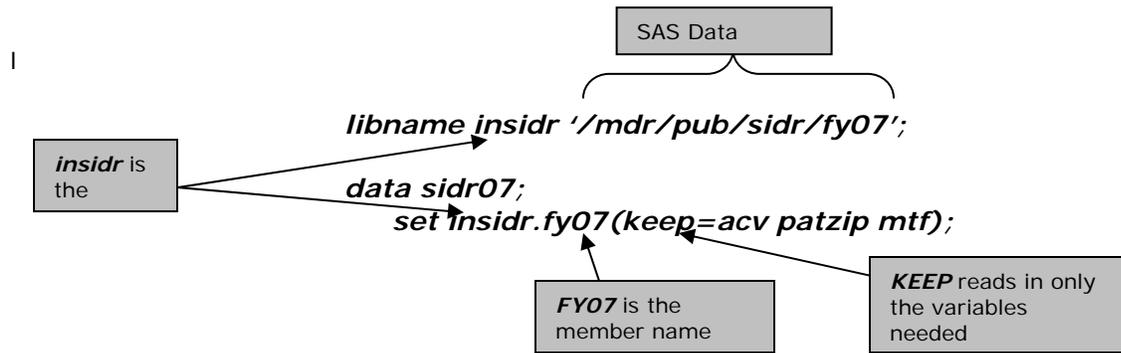
The statement that follows the 'data' statement differs depending on the type of file being read in. The dataset name ('dsname') should be something intuitive, as you may need to refer to the label later.

SAS Data Libraries and Members

It is very easy to input a SAS dataset. SAS already knows the formats of all of the variables in a SAS dataset, so all there is to do is to tell the computer the library reference and member names to be read in.

The following code will read in three data elements in the first 100 observations of the SAS dataset '/mdr/pub/sidr/fy07/fy07.sas7bdat'.

options nocenter obs=100;



The 'data' statement creates the temporary data set called 'sidr07'.
 The 'set' statement that follows is used with SAS datasets only.

```
data dsname;
set libref.membername(keep=var1 var2 ... varn);
```

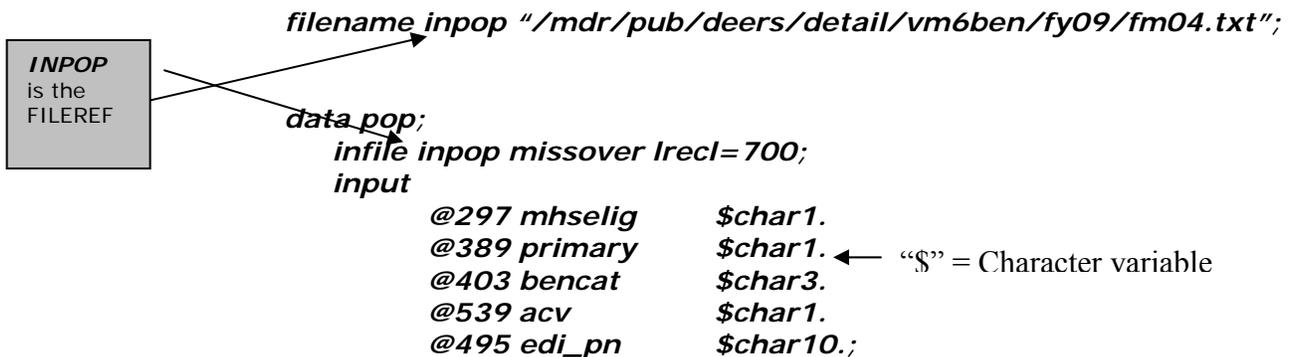
The 'keep' statement is very important. If there is no 'keep' statement (i.e., set insidr.fy07;) then all data elements are read in. This is resource intensive, and will cause protected health information to be read in, even if it is not needed. It should be standard practice to use a 'keep' statement whenever reading in SAS datasets. The variable names used in the 'keep' statement should be those found in either the MDR Functional Specifications or the MDR Data Dictionary. Spelling must be exact. The appendix of programs (Appendix C) contains examples of data steps using all of the major SAS datasets in the MDR. It would be very helpful for MDR beginners to copy and paste from the provided programs and then just modify as needed.

Text Files

Text files are much harder to input than SAS datasets. That is because SAS does not know how the data are organized, what the variables are, or how they are formatted. These are important things for SAS to know in order for it to run properly.

There are two types of text files; fixed length and delimited.

The statement for reading in an uncompressed fixed length file is:



The data step creates a temporary data set called 'pop', just like as when reading a SAS dataset.

'Infile' is used with text files, instead of 'set'. After the 'infile' statement follows the file reference name used in the file declaration section of the program (in this case, 'inpop'). The 'misover' and 'lrecl' components of this statement are options. SAS allows a variety of options with an 'infile' statement. More detailed descriptions of these are described in SAS user manuals.

'Misover' and 'truncover' are two related options that tell SAS how to handle missing values. Without them, if SAS encounters an input line of data with a missing data element, it will skip over the rest of that line of data and go to the next one! The statement "SAS went to a new line when INPUT statement reached past the end of a line" in your SAS log means that you need a 'misover' or 'truncover' parameter. The difference between the two is exceedingly obscure and outside the scope of this document.

The 'lrecl' (logical record length) is used to indicate the length of the input lines in the file you are reading. You can determine the length of input files by looking at the functional specifications. If you do not have an 'lrecl' statement and the data you are reading in goes beyond the default allowed, your SAS log will tell you about it! The error message will say "one or more lines were truncated". You can also assign an 'lrecl' to a longer record length than actual to ensure SAS will not truncate.

The 'input' statement tells SAS that the next lines will specify '@' positions and the types and lengths of data being read in. To determine which '@' position to use in your programs, consult the MDR specifications or data dictionary. Each variable you want to use must be listed, along with its type and format described. The last element to be read in concludes with a semi-colon.

Users get to make up their own variable names with text files. You can call the data elements anything you want! If you do not specify variable lengths, SAS will assign the defaults.

Character variables: Use the '\$charn.' to read in a character variable, where 'n' is the length of the variable.

@20 name \$char20.

In this example, SAS would read in from position 20 ('@20'), 20 characters of data (\$char20.), assigning the label as 'name'.

Integer variables: Specify the length of the variable followed by a '.'. Integer variables can never be less than a length of 3.

@10 days 4.

In this example, SAS would read in from position 10 a field called 'days' that would be an integer value up to 4 digits.

Decimal variables: Use the SAS 'w.d' format. The 'w' specifies the total width of the field, and the 'd' indicates how many digits to read after the decimal point.

@20 costs 10.2

In this example, SAS would read in from position 20, 10 digits of data (for example: 1234567.89). SAS would not read anything past the 10th character.

Date variables: Dates are sometimes simply read in as character. But if you need to calculate the length of time between days, or if you want to use some of the nice SAS date functions, you will need to use special date formats. SAS formats for dates correspond to the ways that you might see dates appear in the real world. These will be discussed at length in the 'formats' section of this document.

Input Statements (Delimited Data)

With fixed width data, the same data elements appear in the same positions on all rows of data. Delimiting data is a technique often used to save storage space when writing out permanent data sets. Delimiters are usually visible when looking at the data (but not when the delimiter is a tab). An example of a delimited line of data is:

```
3!00-04!W4367854!W4367854!ACT
```

The delimiter in this line of data is a '!'. Other common delimiters are pipes ('|'), commas, and tabs.

To read in a delimited data file, the following code works:

```
filename testfile "/hpae2/kennell/linda/uic.txt";  
  
data test;  
  infile testfile dlm='!' missover dsd lrecl=1000;  
  input @1 age  
    agegrp           :$5.  
    assgn            :$10.  
    attach           :$10.  
    bencat           :$3.;
```

The option 'dlim' is used to indicate that the delimiter is an '!'. For tab delimiter data, use "dlim='09'x". You will not see the '09'x if you visually review tab delimited data, but it is there! 'Missover' and 'lrecl' have already been discussed. The 'dsd' (delimiter separated data) option is used to properly read in variables that do not have a period for missing values (i.e., 2 consecutive delimiters).

If the data element being read in is a number (like age), no variable length should be specified. For character variables, use ':n'.

Efficiency and Input Statements

Many of the datasets in the MDR are extremely large. Though the MDR has powerful processors, it is best to subset data while reading it in, if only some records are needed and the dataset being used is large. The method for doing this is different, depending on whether the dataset being read in is a text or SAS dataset. Of course, it is easier to subset input data when using a SAS dataset.

The statements 'keep' and 'drop' can be very helpful in limiting the amount of data read to only what is needed. These statements can be used as parameters when reading in SAS datasets as already shown. 'keep' and 'drop' can also be used after data has been read in. The format is simply 'keep varname1 varname2 etc'.; or 'drop varname1 varname2 etc'.

Conditional Input Statements

The code below reads in the FY07 institutional TED SAS Dataset. When SAS executes this code, it reads in all 6 requested variables only when the 'where' statement is true.

```
libname intedi07 '/mdr/pub/tedi/fy07';

data tedi;
  set intedi07.header(keep=fy acv denrsite adm paid bencat).
    where (acv in ('A', 'B', 'D', 'E', 'F', 'H', 'J', 'M', 'Q') and
    denrsite='0124');
```

Conditional Inputs for Text Files

To limit the amount of data read in from especially large text files, such as the PDTS or PITE, use code such as this:

```
filename inpdts pipe "uncompress -c
/mdr/pub/pdts/detail/fy08/pdts.detail.fy08.txt.Z";
```

```
data pdts;
  infile inpdts lrecl=700 missover;
  input @367 patage 3. @;
  if (patage ge 55) then do;
  input @17 rxnum $char7.
    @31 qty 9.3
    @40 dayssup 3.;
  end;
  else delete;
run;
```

Must have a second INPUT statement when performing a conditional read

Must use the "@" symbol when doing a conditional read.

Only those records meeting the condition are kept.

The only difference is the way that input statement is written. In the example above, PDTS records are needed for patients 55 and older. Since the annual PDTS file contains more than 100 million records, it makes sense to use a conditional read statement. With text files, a conditional read will input only the data elements that are needed to determine if you want the record. If you do want the record, then the remainder of it is read in. This minimizes the amount of data kept in working memory and can be especially important with very large programs. The appendix of programs contains examples of conditional reads that can be customized if needed. Note that the code that is highlighted red is what the user would change about this statement if something different were needed.

The conditional read statement is using some more complex logic (such as the 'do' statement) that will be covered later.

Programming Technique: After Data is Input, What Next?

The process of writing code to create good input data sets has been described. Any good programmer will tell you that the most important part of any program is testing. Good programmers make mistakes all the time. To test input statements, you will use a variety of techniques discussed later in this document. Fundamental to the process is reading in a small number of observations and visually reviewing printed output, and also producing frequency tabulations and basic statistics on key variables.

Functions

SAS is a very robust programming language with nearly infinite possibility for processing data. Once data has been read into memory, you can create new data elements (like 'user defined objects in M2') using SAS functions.

SAS has many functions and in this session, we will address many of the more important ones. Use SAS Manuals for information about more functions – there are literally thousands of them available.

Functions are used in SAS statements that appear after the SAS data is read in (i.e., after the set or input statements).

Math

SAS can be used to do regular math. In this data step, a variable called 'reg_days' is created by subtracting days in the ICU from total bed days. The data element would be numeric of length 8, since not otherwise specified.

```
data sidr;  
  set in1.fy08(keep=mtf dmisdays icudays);  
  reg_days=dmisdays-icudays;
```

Length

The 'length' statement is used to set the length of variables you would like to create. The minimum length for a number is 3. A character can be between 1 and 256 in length.

```
length alos 5.2 bencat $3;
```

Strings

The substring function ('substr') extracts part of a character string. The syntax is:

```
substr(argument,start position,length)  
  
length meprs3 $3 dx_temp $3 year $4 month $2;  
meprs3=substr(meprscd,1,3);  
dx_temp=substr(dx1,1,3);  
year=substr(begdate,1,4);  
month=substr(begdate,5,2);
```

The concatenation function ('||') puts character variables together into one (the opposite of substring). The syntax is:

```
var1||var2||varn
```

```
length yrmn $6;  
year='2003';  
month='02';  
yrmn = month || year;
```

Result: yrmn=200302, as a character, not number, field.

The concatenation operation does not trim blanks. The 'trim' function will do this

```
Firstnm = 'Jane      '  
Lastnm  = 'Doe      '  
fullnm=trim(firstnm)||' '||lastnm;
```

Result: fullnm = Jane Doe (spaces removed)

If Then Else

'If then else' is among the most important language in any programming language. This clause allows users to perform operations on data that meet specific criteria.

The ***syntax*** is:

```
IF expression THEN statement;  
ELSE statement;
```

The 'else' statement is not required, but often needed. When SAS executes 'if then else' statements, if the value of the 'if' expression is true then the statement following 'then' is executed. If an 'else' is present, then the alternate statement is executed.

```
if patsex='F';                /* keeps only records for females */  
if mtf >= '0600' then delete; /* deletes DMISIDs greater than/equal to 0600 */  
  
length agegrp $5;            /* create age groupings */  
if age <= 10 then agegrp='00-10';  
  else if age < 21 then agegrp='11-20';  
  else if age < 66 then agegrp='21-65';  
  else agegrp='65+';
```

Note: 65+ AGEGRP contains any missing AGE values that might have been present.

```
If enc <> 0 then avgrvu=rvu/enc;
```

When using 'if then else' statements, most users will use indentation to keep the clauses separated.

Even though only one statement is allowed between the 'then' and the 'else', a 'do/end' combination can be used to execute multiple lines of code within an 'if then else' statement. To use a 'do/end' statement, simply put the needed lines of codes between the 'do' and the 'end'. For every 'do' there must be an 'end'!

The code example below determines whether or not an encounter record is 'outpatient' and if so, the record is counted as an encounter (enc=1) and total RVUs are calculated. If not, encounters and RVUs are set to 0.

```

if (substr(meprs3cd,1,1)='B' or meprs3cd in ('FBI' 'FBN')) then do;
    enc=1;
    totrvu=workrvu + pervu
end; /* end for substr */
else do;
    enc=0;
    totrvu=0;
end; /* end for else */

```

There are two lines of code between each do and end

You must have a 'end' statement with every 'do' statement!

'If then else' statements and 'do/end' statements can be nested. The method to do this is intuitive, but this can get tricky quickly. If SAS encounters multiple 'do/ends', it will check to be sure that each 'do' has an 'end' (compile) and then process the 'do/ends' from the inside out, much like with multiple parentheses in the arithmetic order of operations. If you need to do this, use comment statements liberally so you know which 'ends' go with which 'dos'. Nested 'if then else' and 'do/end' statements is beyond the scope of this basic SAS document.

Comparison Operators

Comparison operators for SAS are described in the table below.

Table 8: Comparison Operators

Comparison	Operation	Example
Equal to	eq or =	If patsex='F';
Not Equal to	ne or <>	If mhslieg ne 0;
Greater Than	gt or >	If age gt 64;
Greater Than or Equal to	ge or >=	If age ge 65;
Less Than	lt or <	If age lt 66;
Less Than or Equal to	le or <=	If age le 66;
In	in ()	If mtfsvc in ('A' 'F' 'N');
Not In	not in ()	If tnexreg not in ('O' 'A' ' ');

Comparison operators are commonly used with 'if then else' statements.

Formats and the Put Statement

SAS has many format statements available to allow users to change the format or even the content of a data element. This is a broad topic.

SAS itself has some pre-canned formats, and so does the MDR. You can also create formats yourself either in your program or from a text file. SAS recognizes formats in a number of ways. Predefined SAS formats are all described in SAS manuals. MDR formats (available for DMISIDs and Market Area Attributes) are described in functional specifications. Each format has a name.

The syntax for applying a format is:

newvar=put(oldvar,format.);

The MDR DMISID and CAD formats operate similarly. The DMISID format will be described. When the MDR DMISID file is prepared, a format file is also output. This file actually contains executable SAS code that can be 'included' in a program. When the format file is included, the formats within it can be used.

To include the MDR DMISID format file, use a '%include file_name'. For each DMISID, the format contains a very long string of data that holds information about the DMISID. The string of data is in a fixed format so that once the string of data is available, it can be parsed using a substring command to retrieve the attributes of interest.

String from DMISID Format:

```
0001 0001000100010001040404 AS0001Y0001A 0.00 0.00 158.18 137.95 166.35
138.62 0.00 145.92 0.00 0.00 35809ALY 0.9780 74.2000 74.0671190.4164 73.3164
39.6385 74.2583 65.5878 63.6040 85.3918 84.9648 65.2127 64.3248'
```

To determine the positions of the data represented in the string, see the MDR DMISID Specification. Think of the format as a line of data, and by using the 'put' command, you create a variable that contains that string. The code below includes the DMISID format from the MDR, defines a variable to hold the string, and then uses the 'put' statement to assign the contents of the string to the variable 'mtfstring'.

```
%include "/mdr/ref/dmisid.index.fy08.txt"; /* name of format is $par08X. */
/* layout of format in MDR spec */

data temp;
  set inwendy.fy08(keep=enrmtf fullcost);
  length mtfstring $50;
  mtfstring=put(enrmtf,$par08x.); /*returns entire DMISID string */
```

Or you can use the substring command in conjunction with the format to retrieve attributes of interest.

```
length enrsvc $1 enrreg $2 enrreg $2;
enrsvc=substr(put(enrmtf,$par08X.),39,1); /* Branch of service */
enrreg=substr(put(enrmtf,$par08X.),31,2); /* Region - 01, 02, ... */
enrtreg=substr(put(enrmtf,$par08X.),40,1); /* HSSC region - N, S, W, O */
```

After SAS executes the code directly above, the data file 'temp' will contain enrmtf, enrsvc, enrreg, enrreg and fullcost.

Once a format has been loaded (e.g., '%include'), it can be used throughout the program and with any variable that is coded with the expected coding schema, in this case, a DMISID.

```
enrsvc=substr(put(enrmtf,$par08X.),39,1);
mtfsvc=substr(put(mtf,$par08X.),39,1);
```

Both 'enrmtf' and 'mtf' are DMISIDs.

Date Functions and Formats

SAS can store date values as character variables (i.e., encdate=20080501) or as a number that can be used in mathematical calculations. If you read the date in as a character, no math can be done on the date. For that reason, some users simply read in dates in SAS format.

Suppose at position 12, your data file has the following: 19890912

```
input @12 encdate $char8.; /* will keep the character format: 19890912 */
input @12 encdate yymmdd8. /* will translate 19890912 to a SAS date */
```

If the 'yymmdd8.' format is used, the data in your file must be perfectly formatted. For example, if a line of data included a value like '19890431', there would be an error – there is no April 31st! Some users will read date variables in as characters (\$char8.) and then use SAS functions to convert for this reason. Data stored in the MDR is generally clean and this is not usually needed.

SAS dates simply represent the number of days since January 1, 1960. For example, the following calendar date values represent the date July 26, 1989: 072689, 26JUL89, 7/26/89, 26JUL1989, etc. The SAS date value representing July 26, 1989 is 10799. Dates prior to January 1, 1960 are negative numbers and dates after are positive numbers. Using SAS dates allows users to calculate the length of time between two events (for example, length of stay, length of enrollment, length of time between an intervention and an event, etc.).

There are many helpful SAS functions and statements that can be used with dates. The table below summarizes these commands and provides an example of results assuming the variable 'caldate' contains the value '19890912'.

Table 9: Summarized SAS Commands

SAS Syntax	Result
mon=substr(caldate,5,2);	9 (number)
day=substr(caldate,7,2);	12
yr=substr(caldate,1,4);	1989
sasdate=mdy(mon,day,yr);	10847
date1=input(caldate,yymmdd8.);	10847
sasdate="12SEP1989"d;	10847
mon=month(sas date);	9
day=day(sas date);	12
yr=year(sas date);	1989

Procedures

After data is read in and has been manipulated as needed, SAS has a suite of pre-canned procedures that can be employed to further analyze, manage or process data. (You cannot use a SAS procedure in the middle of a data step; the data must first be read in.) These procedures are extremely powerful! Users customize the procedures according to specific SAS rules (parameters and options).

The SAS Manuals describe all available SAS procedures. SAS has a very robust set of statistical tools that will not be discussed in this document. This document focuses on the most commonly used procedures in the MDR. These are:

Table 10: Most Common Procedures Used in MDR

SAS Procedure	Purpose
Contents	Information about a SAS dataset, (temp or permanent)
Print	Print output

SAS Procedure	Purpose
Sort	Sorts a dataset
Freq	Crosstabs, frequencies and basic statistics
Summary	Summarizes data
Format	Creates formats
Datasets	Mgmt of datasets in temporary memory

The word 'proc' is reserved in SAS to represent the use of a procedure.

The basic syntax is:

proc *procname*;

Unless otherwise specified, the procedure will operate on the most recent data in memory. Using the option 'data=' allows the user to override that default. This option is commonly used.

proc *procname* data=*dsname*;

Contents

'Proc contents' prints information about the contents of libraries and data sets to the SAS list file (*.lst'). It provides general information about the size, variables, member names, formats, etc. of data within the libraries / datasets. This procedure does **not** print out the exact contents of the file (i.e., all the records in the file) but provides information about the SAS data set. This procedure can be used with permanent data files from the MDR, for example, or with datasets in temporary memory.

Code for 'proc contents' of MDR SIDR File for FY08:

```
options nocenter;
libname insidr08 '/mdr/pub/sidr/fy08/sidr.fy08';

proc contents data=insidr08.fy08;
```

The process to use to run a proc contents on an MDR file is:

1. Open PICO at the command line (pico contents.sas)
2. Type in program (above)
3. Hit [cntl X] to save program
4. Run program using the SAS command (sas contents.sas³)
5. Review the SAS Log (more contents.log)
6. Review the output from the procedure (more contents.lst)

You will follow this process again and again in the MDR!

The list file contains the output from 'proc contents'.

At command prompt type: more contents.lst

An example is:

³ Refer to the MDR User's Guide for command syntax when saving logs for HIPAA requirement.

```

Data Set Name: INSIDR08.FY08           Observations: 180941
Member Type:   DATA                   Variables:   247
Engine:        V8                       Indexes:    0

Created:       12:33 Monday, April 14, 2008   Observation Length: 1688
Last Modified: 12:33 Monday, April 14, 2008   Deleted Observations: 0
Protection:                               Compressed: NO
Data Set Type:                               Sorted: NO
Label:
  
```

-----Engine/Host Dependent Information-----

```

Data Set Page Size:      65536
Number of Data Set Pages: 5789
First Data Page:        1
Max Obs per Page:       38
Obs in First Data Page: 20
Number of Data Set Repairs: 0
File Name:               /mdr/pub/sidr/fy08/sidr.fy08/fy08.sas7bdat
Release Created:         8.0202M0
Host Created:            AIX
Inode Number:           338434
Access Permission:      rw-rw----
  Owner Name:            madm
  File Size (bytes):     379396096
  
```

-----Alphabetic List of Variables and Attributes-----

#	Variable	Type	Len	Pos	Label
142	ACV	Char	1	883	DEERS ALTERNATE CARE VALUE
245	ACV2	Char	1	1646	DERIVED ALTERNATE CARE VALUE
17	ADMDATE	Num	8	16	ADMISSION DATE, SAS DATE
15	ADMSRC	Char	1	311	ADMISSION SOURCE
20	AUTOPSY	Char	1	314	AUTOPSY INDICATOR
ETC.					

It is common to want information about all datasets within a SAS library at once. The `'_all_'` option does this. When using this option, the list file will contain separate information for each member in the library.

```

options nocenter;
libname insidr08 '/mdr/pub/sidr/fy08/sidr.fy08';

proc contents data=insidr08._all_;
  
```

'Proc contents' is also used within computer programs, to get information about temporary datasets. This is useful when debugging, or when output must be in a specific format.

```

options nocenter;
libname insidr08 '/mdr/pub/sidr/fy08/sidr.fy08';

data inpat;
  set insidr08.fy08(keep=mtf fullcost);
  disp=1;
  avgcost=fullcost/disp;
run;

proc contents data=inpat;
  
```

run;

The contents would show the temporary dataset 'inpat' contains four variables: mtf, fullcost, disp and avgcost. The 'data=' option can also be used with 'proc contents'.

Print

'Proc print' prints the observations in a SAS data set. 'Proc contents' gives information about the dataset itself, but 'proc print' will print all observations from the most recent dataset in memory to the list file. Options can be used to limit the number of observations and variables printed, and to title the output.

'Proc Print' is excellent for code development and debugging. All good programmers know that there is no substitute for what is often referred to as a 'desk check'. The programmer takes a small number (sometimes specially crafted) of observations and runs them through programming code, step by step, reviewing printouts and comparing with hand calculations. 'Proc print' is the tool used to do this. It is also used for visual inspection of data for formatting errors and such.

The syntax and common options for proc print are:

PROC PRINT <option-list>;

VAR- Prints specified variables in the order listed. If you do not use a VAR statement, by default, all the variables will be printed.

TITLE – Prints title in output.

DATA= - Specifies dataset to be printed. If you do not use a DATA statement then the most recent dataset in memory will be printed. Can also use an (obs=n) parameter as a data option.

Example of code for proc print;

```
libname in1 '/hpa2/kennell/wendy';  
options nocenter ps=50;  
  
data sidr;  
    set in1.outputpat(keep=dx1-dx10 asthma patuniq prn mtf);  
run;  
  
proc print data=sidr(obs=50);  
    var dx1-dx10 asthma;  
    title 'QC: sample print out of records';  
run;
```

The SAS list file would contain 50 observations of data from the SAS dataset 'sidr'. There would be 11 variables printed (10 diagnosis codes; dx1- dx10, and asthma).

Even though the dataset contains patuniq, prn and mtf (they are referenced in the keep statement), only the requested variables will print.

Sort

'Proc sort' sorts the observations in the most recent data set in memory by one or more user specified variables or a data set of your choosing using the 'date =' option. Results replace the input data set, but an option can be used to save separately if needed.

Sorting is among the most resource intensive processes that a computer can perform. Only sort when needed.

One reason sorts are needed is when SAS requires sorted input to do a procedure or to combine data. In some cases, there are alternative techniques, and those will be taught here. There are situations where sorting cannot be avoided, though.

'Proc Sort' uses ASCII sorting order (smallest to largest):

```
blank ! " # $ % ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B C D E F G H I J
K L M N O P Q R S T U V W X Y Z [ \ ] ^ _ ` a b c d e f g h i j k l m n o p q r s t u v
w x y z { | } ~
```

The syntax and common options for proc sort are:

```
PROC SORT <option-list> <collating-sequence option>;
BY <DESCENDING> variable-1 ..... <DESCENDING> variable-
n;
DATA=
OUT=
NODUP
NODUPKEY
```

The "by" option is used to specify the variables to sort by. Adding the word 'descending' will sort in reverse order (the default is ascending).

```
data drgs;
    set in1.mtf(keep=mtf drg disp);
run;

data costs;
    set in2.mtf(keep=mtf drg fullcost);
run;

/* sorts 'costs' by drg since costs is most recent in memory */
proc sort data=costs; by drg;
run;

/* data= option overrides default */
proc sort data=drgs; by descending disp; run;
```

The 'out' option creates a new temporary dataset to store the sorted data, instead of overwriting the input data. Be careful with this option when using very large datasets. There are memory limits and programs will bomb if they run out of memory!

The 'nodup' statement will eliminate observations that are complete duplicates of one another.

```
proc sort nodup;  
    by drg;  
run;
```

The 'nodupkey' statement will keep only one observation when 2 observations with the same combination of 'by variables' are encountered.

```
proc sort nodupkey;  
    by drg;  
run;
```

'Proc sort' can be used with no options, some options or all of the option at once. For example:

```
proc sort data=sumtedi(keep=drg adm paid) out=outsum;  
by descending paid;  
run;
```

Freq

'Proc freq' (usually pronounced "freak") produces one way frequencies and cross-tabulations, and produces associated statistics. Users specify which variables 'proc freq' should analyze. 'Proc freq' prints results to the SAS list file, but an option can be specified to write to a temporary SAS dataset.

This procedure is very commonly used to produce frequencies and cross-tabs. It is also sometimes used to look at relationships between variables, and other interesting statistical variables. This document focuses on the most frequently used options for frequencies and cross-tabs. 'Proc freq' is extremely flexible. Consult the SAS manual for more options.

The syntax and common options for proc freq are:

```
PROC FREQ <option-list>;  
    TABLES <var> <var1*var2> /  
    <Missing>  
    <List>;
```

The 'tables' statement specifies which variables to tabulate. This can be one or more variables, and can include cross-tabulations.

```
proc freq;  
    tables drg;  
  
proc freq;  
    tables drg*recbenf;
```

When doing more than 1-way tabulations, it is recommended that you suppress the default output and instead use 'list' format. 'List' format is easier to read and can be easily used in spreadsheet software. The code to do this is:

```
proc freq;  
    tables drg*recbenf/list;
```

If the 'missing' option is not used, records with missing values in the variables being analyzed will be discarded. This is usually not desired, such that the 'missing' option is usually used.

```
proc freq;  
tables drg*recbenf/list missing;
```

When the output from a frequency table is needed for later use, the 'out' option will write the results (variables, count and other key information) to the dataset specified in the out=option. The 'noprint' option suppresses the output being written to the *.lst file.

```
proc freq noprint data=insidr;  
tables drg*recbenf / list missing out=sidrstats;
```

When SAS executes this proc freq, a temporary dataset will be created called "sidrstats" containing drg, recbenf, count and other statistics.

Summary

'Proc summary' is among the most commonly used procedures in SAS. This procedure behaves much like M2, aggregating data based on user-specified strata. 'Proc Summary' outputs the tabulated data to a new SAS dataset (can be either temporary or permanent).

'Proc summary' is nearly identical to 'proc means'. A major difference between the two is that 'proc summary', by default, produces no printed output, while 'proc means' does.

With a 'proc summary', either a 'by' statement or a 'class' statement is used to specify which variables to use to stratify the data. The 'by' statement requires that the input dataset be sorted in order of the 'by' variables. Because sorting is very resource intensive, many programmers use the 'class' statement instead. The 'class' statement, with the 'nway' parameter produces output equivalent to that with a 'by' statement, but without requiring a sort.

The syntax and common options for 'proc summary' with a 'class' statement are:

```
PROC SUMMARY NWAY <option-list>;  
CLASS var1 ..... var n / MISSING;  
VAR numvar1 .... Numvarn;  
OUT=DSNAME <list of statistics and var names>;
```

Example of common code with 'proc summary': This program tabulates population by gender.

```
proc summary nway data=temp(keep=dmissex pop);  
class dmissex / missing;  
var pop;  
output out=outpop sum=pop;
```

'Proc summary' does not automatically print output, but it does prepare a dataset, in this case 'outpop' which can be used later in the program. This is a very commonly used feature. Consider an example where you need to tabulate person-level statistics across multiple sources of data (i.e., total cost = mtf inpat \$ + mtf outpat \$ + psc inpat \$, etc.).

To do this type of task, you must summarize data at person level in each source and then tie together (later).

To print output from a 'proc summary', use 'proc print'.

```
proc print data=outpop;  
var dmissex pop;
```

At command prompt type: more popsum.lst

Output: Proc Summary with Class statement.

obs	gender	pop
1	F	720
2	M	704

'var' Statement: Identifies the analysis variables and their order in the results
If you omit the 'var' statement, then 'proc summary' produces a simple count of the observations.

'output' Statement: Creates an output data set that contains the specified statistics and identification variables.

Statistics: Many statistics can be produced in 'proc summary', including minimums, maximums, averages, standard deviations and record counts, among others. Consult your SAS manual for additional information.

Example of Proc Summary with more statistics:

```
proc summary nway;  
class mdr / missing;  
var beddays  
output out=sumout n=count sum=days average=alos;
```

Datasets

'Proc datasets' provides data management functions. This procedure retrieves information about the SAS data library or datasets, such as the names of files, the size of each file. 'Proc datasets' allows you to list, copy, append, rename and delete SAS files in the SAS data library, allowing you to better manage your library by freeing up or saving space.

The most common use of 'proc datasets' is indeed to free up space. After using a temporary dataset, if it is no longer needed, it can be deleted from working memory, allowing SAS more space to process the rest of the program. This procedure is often necessary when working with multiple years of data and large datasets.

```
libname insidr08 '/mdr/pub/sidr/fy08/sidr.fy08';  
  
data a;  
set insidr08.fy08; /* All FY08 records */  
run;  
  
data b;  
set a;
```

```

        if mtf='0124'; /* Records for site 0124 only */
run;

proc datasets nolist;
    delete a;
run;

```

This program removes the temporary dataset "a" from memory.

Format

'Proc format' is a handy procedure for assigning labels to SAS data. The pre-canned MDR DMISID 'proc format' and its application have already been discussed. This procedure format allows users to create their own formats.

```

PROC FORMAT;
    VALUE $fmtname
        Value 1 = 'string1'
        Value 2 = 'string2'

        Value N = 'stringn'
        Other='Default';

```

The values are a list of expected values to be found in a data column, and the strings represent the data to be assigned (applied) to the value in the dataset.

One common way 'proc format' is used is to create outputs with user friendly descriptions. The following code creates a format called 'gender' which is then applied in the program to the variable 'patsex'.

```

options nocenter;
libname tempdat '/hpae2/kennell/wendy/temp';

proc format;
    value $gender
        'F' = 'Female'
        'M' = 'Male'
        Other='Unknown';
run;

data a;
    set tempdat.fy08(keep=patsex fullcost);
    gender_desc=put(patsex,$gender.);
run;

proc print data=a;
    var gender_desc fullcost;
run;

```

This format can be applied to any variable in the program with the F/M coding schema. This concept is especially useful when considering the DMISID format. Many MDR datasets contain numerous variables coded with a DMISID. The MDR DMISID Format allows users to

create attributes of the DMISID variables easily, simply by using different variables in the put statement.

'Proc formats' can be typed in by hand if the lists of values are small. If the lists of values are large, a program can be used to create the 'proc format'. An example is included in the appendix with example programs.

Combining Data Sets

Often the combination of data from different files or from output created in different parts of a program are needed to answer a question. SAS has a variety of techniques for combining data. For M2 users, much of this section will be similar conceptually to unions, subqueries, intersections, etc.

Appending Datasets

To append two data sets together, use the 'set' statement. The 'set' statement has already been discussed in the context of inputting SAS datasets. It can also be used to append one dataset to another.

Code for appending datasets from the same SAS library

```
libname sidrsum '/hpae2/kennell/wendy/sidr';
options nocenter obs=max;

data a;
    set sidrsum.fy08(keep=fy mtf fullcost)
        sidrsum.fy07(keep=fy mtf fullcost);
run;
```

The members 'fy07' and 'fy08' from the 'sidrsum' SAS library are appended to one another in a temporary dataset called 'a'.

Code for appending temporary datasets in SAS

```
data direct;
    set in1.fy08(keep=patuniq rwp);
run;

data psc;
    set in2.fy08(keep=patuniq rwp);
run;

data both;
    set direct psc;
run;
```

The dataset 'both' would contain all records in both 'direct' and 'psc'.

When appending datasets together, if there are variables in one file that are not in the other, the resulting SAS dataset will contain the superset of data elements (any data element in either input dataset), with blank values for the missing data. Sometimes further processing is required as a result.

Illustration of SAS treatment of non-overlapping variables when appending datasets.

Dataset A		
OBS	Type	Var1
1	X	A
2	Y	B
3	Z	C

Dataset B		
OBS	Type	Var2
1	X	A1
2	Y	B1
3	Z	C1

Dataset Combine			
OBS	Type	Var1	Var2
1	X	A	
2	Y	B	
3	Z	C	
4	X		A1
5	Y		B1
6	Z		C1

} Var2 is missing in the first 3 observations because Var2 is not in the input dataset A.

Note that the first three observations from the data set (combine) are from dataset A and the last 3 from dataset B.

Merging Datasets

The 'merge' statement in SAS is used to join two or more datasets together by merging observations from each into a single value. This technique is used to union or intersect datasets.

```
DATA DSNAME;
  MERGE DS1 (in=a) DS2(in=b) ..... DSn(in=X);
  By var1 var2 ...etc;
```

In order to create a merged dataset, all input datasets **must be sorted** by the 'by' variables in the sort. Since sorting is resource intensive, sometimes workaround techniques are used to avoid a sort.

Another requirement for merging datasets using the 'merge' statement is that at least one of the datasets must contain only one row of data for each combination of the by variables. The 'merge' statement works with one-to-one merges and one-to-many merges. Many-to-many merges are best done with a 'proc sql'. This technique is outside the scope of this document.

When SAS executes the 'merge' statement, the variables after the 'in=' statement (a, b, etc.) are logical operators that can be used to selectively delete records from the merged dataset.

In the following example, a direct and purchased care dataset are merged by person identifier. Records are retained only if the person identifier is found in both data set 'direct' (corresponding to 'a') and 'purchased' (corresponding to 'b'):

```
data both;
  merge direct(in=a) purchased(in=b);
  by patuniq;
```

```

if a and b;
run;

```

This technique is usually referred to as an intersection or an inner join. Records can be included using an 'if' statement according to the following table:

Table 11: Including Records Using 'If' Statement

In=	Keeps records	Example Use
a and b	Both datasets	Continuous eligibility check, users of both direct (a) and purchased care (b)
a or b	Either dataset	Beneficiaries eligible for direct or purchased care, beneficiaries diagnosed with diabetes in either direct (a) or purchased care (b)
a and not b	In dataset a and not in dataset b	Beneficiaries with diabetes (a) w/o a hbA1C exam (b)
not a and b	Not in data set a but in b	Enrollees in the ER (b) who have not been treated by their PCM in the last 3 months (a)

Using the two datasets below, the conditions from the table are illustrated. Each of the datasets includes costs as person level. The list of persons in each dataset is slightly different.

Direct	Purchased																
<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th>PERS</th> <th>DCCOST</th> </tr> </thead> <tbody> <tr><td style="text-align: center;">1</td><td style="text-align: center;">3400</td></tr> <tr><td style="text-align: center;">2</td><td style="text-align: center;">450</td></tr> <tr><td style="text-align: center;">4</td><td style="text-align: center;">230</td></tr> </tbody> </table>	PERS	DCCOST	1	3400	2	450	4	230	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th>PERS</th> <th>PCCOST</th> </tr> </thead> <tbody> <tr><td style="text-align: center;">1</td><td style="text-align: center;">200</td></tr> <tr><td style="text-align: center;">3</td><td style="text-align: center;">54300</td></tr> <tr><td style="text-align: center;">4</td><td style="text-align: center;">560</td></tr> </tbody> </table>	PERS	PCCOST	1	200	3	54300	4	560
PERS	DCCOST																
1	3400																
2	450																
4	230																
PERS	PCCOST																
1	200																
3	54300																
4	560																

	A or B		A and B		A and not B		not A and B	
PERS	DCCOST	PCCOST	DCCOST	PCCOST	DCCOST	PCCOST	DCCOST	PCCOST
1	3400	200	3400	200				
2	450	.			450	.		
3	.	54300						54300
4	230	560	230	560				

Note that when a variable is missing in a 'merge' statement, SAS stores a '.' instead of a 0 (zero). For example, with the 'a or b' observation, the value for person 2 of PCCOST is missing (indicated by a '.' - it is missing because person 2 is not in the PC file). In order to do math on numeric variables with missing values, you must reset missing values to 0.

```

If pccost eq . then pccost=0;

```

Practical example:

Suppose you wish to categorize users by whether or not they have had a direct care E&M visit, a purchased care E&M visit, both, or neither. After tabulating the direct care encounter (SADR) and purchased care non-institutional (TED-N) records by person ID, a merge would be conducted. The 'a' and 'b' statements can be used with the 'if' statement to create a usertype variable.

```

data people; /* list of eligibles */

```

```

        set in1.elg(keep=patuniq);
run;

proc sort data=people; by patuniq;
run;

data direct; /* list of persons with E&M visit in direct care */
        set in2.mtf(keep=patuniq nummtf);
run;

proc sort data=direct; by patuniq;
run;

data psc; /* list of persons with E&M visit in private sector */
        set in3.psc(keep=patunq numpsc);
run;

proc sort data=psc; by patuniq;
run;

data all;
        merge people (in=a) direct (in=b) psc (in=c);
        by patuniq;

        length usertype $8;
        if b and not c then usertype='mtf only';
        if not b and c then usertype='psc only';
        if b and c then usertype='both  ';
        if a and not b and not c then usertype='non user';
run;

proc freq data=all; tables usertype/list; /* count people by user type */
run;

```

Arrays

Most robust programming languages include a mechanism for processing groups of similar data in the same way, called an array. Think of an array as a convenient way to do exactly the same thing to a group of variables, without having to spell out instructions individually. A good example of the utility of array processing is using diagnosis and procedure codes in health care data. Rather than writing 20 times "if diagnosis code is diabetes", an array allows a programmer to conveniently check all 20 rows at once! Nice shortcut!

The concept of array processing is incredibly complex and powerful. In this document, we focus only on the simplest case, one-dimensional arrays.

An array is a data structure that temporarily organizes groups of SAS variables for easier use. Arrays are identified by an **array-name**. Think of an array as a convenient way to reference a group of variables within your programs. In order to use an array in a program, the array must be declared.

Syntax for array declaration is:

array array_name(n) [\$] list of data elements

array_name can be any meaningful name
(n) dimension of the array, can use () or { }
\$ indicate the array data element is character

Examples:

```
array x_proc(4) $ proc1-proc4;    ← can list proc1-proc4 individually
array x_dx(20) $ dx1-dx20;
array x_sprocd(4) $ sprocd1- sprocd4;

array x_rvu(4) rvu1-rvu4;
```

Arrays are declared within 'data' steps, after data has been read in. Arrays have very specific rules for use! Some rules for arrays processing are:

- The user specifies which data elements go into an array. You can not mix character and numbers in an array. The array must be entirely character, or entirely numeric. If character, use the \$ in the declaration statement as in the examples above.
- The data you place in an array must be consistent with the array definition. (e.g., do not put character data in a numeric array, or vice versa. SAS will not let you!)
- Use length statements to pre-define formats prior to the array declaration.

Example: **length tot1-tot5 6;**
array tots(5) tot1-tot5;

With the length statement, the variables tot1-tot5 will be numeric with 6 digits. If the length statement had not been used, SAS would consider the data numeric, length 8. Length statements can save space.

- Arrays are unique to a data step. You must re-declare arrays as you pass through from data step to data step. (The error message when violating this rule is: undeclared array reference)

Array elements (the variables in the array) are referenced using an index. The index tells you the position you are trying to reference. E.g., an index of 3 would reference the third variable in the array.

Consider the following example data set:

rvu1	rvu2	rvu3	rvu4
1.7	0.02	1.1	2.3

```
array x_rvu(4) rvu1-rvu4;
```

Variable x_rvu(1) has a value 1.7.

Variable x_rvu(2) has a value 0.02... and so on.

Arrays are usually processed with 'do loops'. The structure for a 'do loop' is:

```
do i=1 to n;
```

SAS statements;

end;

When SAS executes a 'do loop', it reads in the line of code, starts with i=1 (or whatever the code tells it to start at), and processes until it hits the 'end'. Once it hits the end, it pops back up to the 'do' statement, increments the i by 1 (default value, can change this if need be, can always work backwards, or in bigger steps) and processes again. SAS continues this until the i reaches 'n', at which point it continues on with the next line of code.

Example 1 – flag PTSD SADR encounters

```
length ptsd $1;           /* Set length of variable PTSD */
array x_dx(4) $ icd1-icd4; /* Declaring array */

ptsd='N';                 /* Initialize variable PTSD */

do i=1 to 4;              /* Loop through each DX in array – flag PTSD */
    if substr(x_dx(i),1,5)='30981' then ptsd='Y';
end;
```

Obs Num	ICD1	ICD2	ICD3	ICD4	PTSD
1	30981	AAAAA			Y
2	BBBBB	AAAAA			N
3	BBBBB	30981	CCCCC		Y

Only one variable was created in Example 1. In Example 2, several variables are created at once, using an array to read data and a different array to hold the newly created data elements. While the array that holds the new data elements will disappear at the end of the data step, the variables that were held in the array stay in memory along with the rest of the dataset.

Example 2 – flag TFL claims based on special processing codes

```
array x_spec(4) $ sprocd1-sprocd4;
array x_tfl(4) $ tfl1-tfl4;

do i=1 to 4;
    if x_spec(i) in ('FF','FG','FS') then x_tfl(i)='Y';
    else x_tfl(i)='N';
end;
```

Obs Num	SPROCD1	SPROCD2	SPROCD3	SPROCD4	TFL1	TFL2	TFL3	TFL4
1	Z	W	RT		N	N	N	N
2	W	FF			N	Y	N	N
3	FS				Y	N	N	N

In this case, the array x_tfl(i) contains the variables tfl1, tfl2, tfl3 and tfl4.

- It is good practice to indent the SAS statements inside a 'do loop', as it makes the code more easily readable, as well as makes it easier to debug.
- The 'i' variable should be dropped when you are done with it.

- The most common error with 'do loops' is forgetting to use the 'end' statements. You must end every 'do loop' with an 'end' statement!

Macros

The SAS Macro language is designed to eliminate repetitive code. SAS Macros are very powerful compared with other tools, such as arrays. Macros can be used to create data sets, perform procedures, etc. A macro is a set of SAS code. The code is usually generic in nature. The SAS language itself includes some very simple macros that can be used at any time. SAS also allows users to write their own macros. This is incredibly powerful!

'%macro_name' indicates the call of a macro program, and '&var_name' indicates a macro variable (one in which substitution will occur).

Available SAS Macros

The most commonly used macros available to all SAS users are '%let' and '%include'.

'%let' is used to assign a value to a global variable, something that can be used throughout a program.

Syntax: **%let** var_name=some_value;

The '%let' statement usually occurs right after the file declaration and options sections of a program. It allows the program to assign the value 'some_value' anywhere the variable &var_name is referenced. Variables where substitution will occur are known as 'parameters'.

Sample code:

```
%let fy2='08';
%let dir='jan09';

libname in1 "/hpa2/kennell/linda/&dir./data";      ←-- replace '&dir' with 'jan09'
                                                    /hpa2/kennell/linda/jan09/data

libname intedi "/mdr/pub/tedi/fy&fy2.";              ←-- replace '&fy2' with '08'
                                                    /mdr/pub/tedi/fy08

data temp;
  set in1.header;
  infile_fy="FY&fy2.";                              ←- replace '&fy2 with 08"
run;                                                      infile_fy="FY08";
```

'%include' inserts the contents of an external file into a SAS program. Usually the external file contains SAS code. It is very common practice to store complex user –defined macros in external files, and use a '%include' statement to incorporate the code when needed.

Syntax: **%include** "filename";

In order for a '%include' to work properly, the contents of the file being included must be well-understood. If the file includes the SAS code, the placement of the '%include' statement must flow logically with the rest of the program.

Sample code:

```
%include "/hpae2/kennell/wendy/macro_temp.sas";
```

User Defined Macros

SAS Macros can be thought of as a 'home-grown' procedure. A programmer can construct SAS macros, which are later 'called'. A SAS macro contains generic SAS code. The concept is similar to that of an algebraic expression. In algebra, an equation might be 'y=x+2'. Entering different values of x into the equation yields different results for y, but the underlying mathematical operation, adding 2 to a number, is the same. With a macro, the underlying SAS code is the equivalent of the equation, and the calling of the macro is the equivalent of substituting a value for x, to get the result y.

Basic Syntax to create a macro:

```
%macro macro_name(parameter1,parameter2, ....parametern);
```

```
SAS statements;
```

```
%mend macro_name;
```

Syntax to call a macro:

```
%macro_name(parameter1=value1,parameter2=value2,....parametern=valuen);
```

Parameters are not required in a macro. When parameters are not needed, the syntax simplifies to exclude the parenthetical notations of parameters and values to substitute for them.

Macro code is not executed until a macro call appears in a program. By calling a macro, the program is requesting SAS to execute all code contained within the macro, substituting the values in the call for the parameters in the macro.

Macro code is essentially the same as regular SAS code, except that when using a variable where you want substitution to occur (a parameter), the parameter name is preceded by an '&' and ends with a '.'.

The example below is commonplace. A user needs the same code to be executed on different files. In this case, the files used are the MDR SIDRs. The program reads in three years of SIDR files and retains records for active duty family members (combenf=1; see DD). It then calculates a frequency of records (i.e., number of dispositions) by MTF.

When a macro is not used, the code is essentially repeated 3 times. When using a macro, the code is written generically, with the year of data as a substitution variable. When executing the code, SAS will skip the code between macro/mend (except to compile it) until the macro is called. After the first macro call, SAS will substitute '06' everywhere the macro parameter ('&fy2') is encountered in the code. Then the code will be executed. When the code from the first macro call is complete, SAS will go to the next statement, which is another macro call, etc.

Code using macro

```

%macro read_sidr(fy2);
  libname insidr "/mdr/pub/sidr/fy&fy2.";

  data sidr&fy2.;
    set insidr.fy&fy2.(keep=combenf mtf);
    where combenf='1';
  run;

  proc freq data= sidr&fy2.;
    tables mtf/missing list;
    title "FY&fy2. SIDR";
  run;
%mend read_sidr;

%read_sidr(fy2=06)
%read_sidr(fy2=07)
%read_sidr(fy2=08)

```

Code not using macro

```

libname insidr '/mdr/pub/sidr/fy06';
libname insidr '/mdr/pub/sidr/fy07';
libname insidr '/mdr/pub/sidr/fy08';

data sidr06;
  set insidr.fy06(keep=combenf mtf);
  where combenf='1';
run;

proc freq data= sidr06.;
  tables mtf/missing list;
  title "FY06 SIDR";
run;

data sidr07;
  set insidr.fy07(keep=combenf mtf);
  where combenf='1';
run;

proc freq data= sidr07.;
  tables mtf/missing list;
  title "FY07 SIDR";
run;

data sidr08;
  set insidr.fy08(keep=combenf mtf);
  where combenf='1';
run;

proc freq data= sidr08.;
  tables mtf/missing list;
  title "FY08 SIDR";
run;

```

A second example shows a SAS macro with more than one parameter. The example is one that is commonly used with population data. Since the MDR stores eligibility and enrollment data in monthly files, it is very common for users to need to read in more than one month of data at a time.

The macro code below reads in the first 6 months of the FY08 DEERS eligibility files. Each monthly DEERS file is stored in a separate directory under a different filename. You could write 6 individual data steps reading each DEERS file individually, and then appending. Or use a macro.

Macro solution:

```

%macro read_vm6(fy2,fm2);

  filename invm6 pipe "uncompress -c /mdr/pub/deers/detail/vm6ben/fy&fy2/fm&fm2..txt";

  data deers&fy2.&fm2.;
    infile invm6 missover lrecl=700;
    input @297 mhselig $char1.
          @307 primary $char1. @;
    if mhselig='1' and primary='1' then do;
      input @539 acv $char4.;
    end;
    else delete;

```

Use the
%macro and
%mend
statements
together!

```

run;

proc append data= deers&fy2.&fm2. base=all_deers;
run;

proc datasets nolist;
    delete deers&fy2.&fm2.;
run;

%mend read_vm6;

%read_vm6(fy2=08,fm2=01)
%read_vm6(fy2=08,fm2=02)
%read_vm6(fy2=08,fm2=03) ←-- when calling macros, do not have to include ';' at the end
%read_vm6(fy2=08,fm2=04)
%read_vm6(fy2=08,fm2=05)
%read_vm6(fy2=08,fm2=06)

```

Here is what the macro is doing:

1. The macro in this code is named READ_VM6. SAS will compile the macro as soon as it sees the '%' sign, but it will not execute the macro until the call statement(s) appear(s).
2. In the 1st macro call **%read_vm6(fy2=08,fm2=01)** , whenever the line '&fy2' is seen, it will be replaced by the parameter in the call statement (in this case '08'). Whenever the line '&fm2' is seen, it will be replaced by the parameter in the call statement (in this case '01').

```
filename invm6 pipe "uncompress -c /mdr/pub/deers/detail/vm6ben/fy08/fm01.txt";
```

```
data deers0801;
    infile invm6 misover lrecl=700;
    input @297 mhselig $char1.
    @307 primary $char1. @;
    if mhselig='1' and primary='1' then do;
        input @539 acv $char4.;
    end;
    else delete;
run;
```

```
proc append data= deers0801 base=all_deers;
run;
```

```
proc datasets nolist;
    delete deers0801;
run;
```

3. In the 2nd macro call **%read_vm6(fy2=08,fm2=02)** , whenever the line '&fy2' is seen, it will be replaced by the parameter in the call statement (in this case '08'). Whenever the line '&fm2' is seen, it will be replaced by the parameter in the call statement (in this case '02').

```
filename invm6 pipe "uncompress -c /mdr/pub/deers/detail/vm6ben/fy08/fm02.txt";
```

```
data deers0802;
    infile invm6 misover lrecl=700;
    input @297 mhselig $char1.
    @307 primary $char1. @;
    if mhselig='1' and primary='1' then do;
        input @539 acv $char4.;
    end;
    else delete;
```

```

run;

proc append data= deers0802 base=all_deers;
run;

proc datasets nolist;
delete deers0802;
run;

```

AND SO ON....

The macro automates the process of reading in data.

'Proc append' is not particularly useful, except for in this type of process. This procedure appends datasets to one another. The 'base=' option specifies the final dataset name that will hold the appended data, while the 'data=' option is used to specify the dataset to append. After the first macro iteration, the dataset 'all_deers' would only contain data from the 'deers0801' dataset (containing fy08, fm01 data). After the second iteration, SAS would append the dataset 'deers0802' to the dataset 'all_deers', so that 'all_deers' would then contain 2 months of data. This process continues with each macro call so that at the end of the macro calls, all_deers contains all months of DEERS data read in.

'Proc datasets' is used to delete the interim data files once they are no longer needed. For example, once 'deers0802' has been added to the dataset 'all_deers', the extra copy is no longer needed in memory. The datasets procedure will get rid of the specified datasets, making room for the next month to be read in.

Output Data Sets

SAS has a number of ways of creating outputs. SAS will commonly write output to a SAS list file, either because it encountered a 'proc print', or because another procedure was used that results in written output (i.e., 'proc contents'). But sometimes, actual data needs to be saved.

Unless otherwise specified, the SAS data step automatically creates a *temporary* SAS data set that is only available during the execution of the SAS program it is created in. This temporary data file cannot be used in another program or downloaded unless permanently saved. When the program is done, data in the SAS dataset is gone.

Data can be permanently saved by writing to an external file. The data in the external files can be viewed (using the more command), downloaded, used in other SAS programs, etc. Just as SAS can read in SAS files or text files, SAS can also write out SAS files or text files.

Writing SAS Datasets

SAS Data Sets are good to use if the data will be used in another SAS program later on. SAS Data Sets are advantageous to text files because SAS remembers all the characteristics of the data in a SAS dataset, the user only needs to know the names of variables (which can be obtained with a 'proc contents').

To write out a SAS data set, first declare the output file in the file declaration section at the beginning of your program. Then use a data step with the syntax below:

```
data libref.memname;
```

This language will create a new member in the library indicated by libref. The reference to the libref with a member name after it is actually what creates the SAS dataset. SAS datasets can be created outside of data steps as well. One common place to create SAS datasets is as part of output of a procedure.

The code below uses the SAS library 'rvu_proj'. This library is used to read in data (in member cy08) and calculate an average RVU. Two new SAS members are added to the library. One is created in the proc summary. The other is created in a data step.

Directory: /hpae2/kennell/wendy/rvu before program:

At the command prompt type: **ls**

You will see the following file listed:
cy08.sas7bdat

Example Code:

```
libname rvu_proj '/hpae2/kennell/wendy/rvu';
options nocenter obs=max;

data rvu;
    set rvu_proj.cy08(keep=mtf totrvu meprs3 enc fy fm);
    if enc > 0 then avgrvu=totrvu/enc;
    else avgrvu=0;
    keep mtf meprs3 fy fm avgrvu;
run;

proc summary nway;
    class fy fm mtf;
    var enc totrvu;
    output out=rvu_proj.mtfsum(keep=fy fm mtf enc totrvu)
           sum=enc totrvu;
run;

data rvu_proj.avgdat;
    set rvu;
run;
```

At the command prompt type: **ls**

You will see the following files listed:
cy08.sas7bdat mtfsum.sas7bdat avgdat.sas7bdat

Writing Text Files

Text files are good when you need to download a file to view or to use in another application. Writing out text files is very similar to reading them in. Declare the filename in the file declaration section of the program. SAS can write either fixed length or delimited files.

Fixed length files: Writing files works just like reading them, except that a few of the key words are different. 'File' is used instead of 'infile' and 'put' is used instead of 'input'.

Example:

```
filename outfile "/hpa2/kennell/linda/out_fixlen.txt";

data dsname;
  set tempds_to_write;
  << more code if needed >>

  file outfile;
  put @1 var1 fmt.
      .....
      .....
      @x varn fmt.;
run;
```

The 'file' statement indicates the name of the permanent file to write to. The 'put' statement is used once. After it, the program must specify each variable to write out, the start position, and the format of the data element (i.e. \$char8., or 12.2). After completing the list of variables to write, a semi-colon is required.

There are other methods for writing fixed length text files. Consult SAS user manuals for more information.

Example:

```
filename out1 "/hpa2/kennell/enrsum.txt";
options nocenter;

/* Assume enroll is in temporary memory */

data _null_; /* use _null_ to write directly to file; no temp memory */
  set enroll;
  file out1 lrecl=30;
  put @1 mon_yr $char5.
      @6 region $char2.
      @8 enr_ct 8.;
```

At the command prompt type: cd /hpa2/kennell/

At the command prompt type: more enrsum.txt

File enrsum.txt will look like this:

```
OCT070110023
OCT0702957
OCT07033344
OCT070487
```

Delimited Files: Writing delimited files is also similar to reading them. The syntax is:

```
Syntax:
  data dsname;
  set tempds_to_write;
```

```

<< more code if needed >>
file fileref;
retain delim 'd';
put @1 var1 fmt.   delim $1.
.....
.....
      varn fmt.      delim $1.;

```

The 'file' statement indicates the name of the permanent file to write to. The 'put' statement is used once. 'Retain' allows SAS to keep the delimiter in memory from record to record. 'd' indicates the delimiter to use, and is commonly a pipe (|), a comma (,) or an exclamation point (!). With a delimited text file, it is not necessary to state any start position except 1. After completing the list of variables to write, a semi-colon is required.

Example:

```

filename out1 "/hpa2/kennell/enrsum.txt";

data _null_;
  set enroll;
  file out1 lrecl=30;
  retain delim '!';
  put @1
      mon_yr      $char5.      delim $1.
      region      $char2.      delim $1.
      enr_ct      8.;

```

At the command prompt type: cd /hpa2/kennell/
 At the command prompt type: more enrsum.txt

File enrsum.txt will look like this:

```

OCT07!01! 10023
OCT07!02!  957
OCT07!03! 3344
OCT07!04!  87

```

An error that can occur when writing out SAS files is referred to as a "W.D" error. Often when calculating data, it is not known how long the resulting variables will be. For example, bed days may be of length 4 on each individual hospital record, but when you add up all bed days for all hospitals, how long does the variable need to be? Most programmers will err on the side of too large, because if the variable lengths are too small, SAS logs are not very friendly in resolving the issue. If you assign a length to a variable that is not as long as it actually is, you will get the following error: *At least one W.D format was too small for the number to be printed. The decimal may be shifted by the "BEST" format.* The SAS log does not tell you which variable is the problem.

Quality Review

The most important characteristic of a good program is that it be right! You will maximize your chances of getting results right if you practice good discipline in terms of program documentation and QC. This section describes some tips for use in developing code, and in checking the ongoing execution of existing code.

Documenting Programs

Create a standard template at the top of your programs to help keep track of why the program exists and other high-level information. For example:

```
*****,  
* Date Created:                               *,  
* Programmer:                                 *,  
* Date Modified:                             *,  
* Program For:                               *,  
* Purpose:                                   *,  
*****,
```

If you are a contractor, you might consider listing your Data Use Agreement (DUA) number for easy reference should there be any question regarding the access needed for the program.

This block of information is presented in SAS comment form, so that SAS does not attempt to compile the statements inside.

Good programs are liberally documented throughout; especially around complex lines of code. Many programmers will note why filters are applied when subsetting data. Comments are also often used to explain complex logic. Finally, comments are helpful in provide instructions for running macros (i.e., changing macro call statements).

Testing Code on Subsets of Data

When developing code, always start by testing piece at a time, on a small number of observations. Using 'options obs=n' is the easiest way to do this. Write small pieces of code and then review printed output visibly. Visual inspection is one of the best ways to spot errors.

Confirmation of Inputs and Outputs

For text files, it is a good idea to print out a few observations for each file you are reading in. This helps confirm that the variables you are reading in are coded as you would expect. If you are reading in a text file this serves as a check that the layout you were provided with is correct. After writing out files, use the head and tail commands in Unix to view the output to ensure it writes as you expected.

For SAS files, include 'proc contents' to confirm variable names and formats.

Logic Checks

Logic checks are especially important. The easiest way to do this is to carefully craft a small number of records to input and run those examples through the code. Then compare what the code does with what you expect (sometimes by doing hand calculations) to be sure the code is behaving as expected. This process is important enough it is given a name: desk checking.

Track Data Flow

Keep careful track of data as it flows throughout a program. Create a spreadsheet that keeps track of record counts in each data step. Review the numbers and be sure they make sense. Look at trends over time if multiple years of data are used. Sometimes code sets change over time, and the same code does not work on all years of data. This technique is helpful for locating these types of errors.

In programs where some records are deleted upon reading in data, instead of deleting the data, write to a separate data set. This way, deleted records can be reviewed later.

Review Logs

Execution errors are when your program fails to run. Some examples include: lack of space, variable not declared, missing semicolon, invalid statement. You will receive an error message in your log.

Logs also contain warnings and other important information. Logs should ALWAYS be read through! Do searches (ctrl w) on the words 'error' and 'warn' as they will be in the logs if the programs had errors.

Produce Summary Statistics and Frequencies

It is important to create and print out summary statistics and frequencies of key variables. This should be done during your program and at the end. By printing out summaries/frequencies during your program you can track changes you made to variables (check new variable coding, recoding of variables, etc.). This is a good way to test your logic. Your final results will also help track trends and distributions.

For example, if you are creating a new data element called age group, print out a frequency of age*age group and check to see whether the ages falls within the right categories.

Comparative Sources

External sources such as the M2 or previous reports that you have run provide great sanity checks. For instance, if you are working with total population, a simple run on the M2 can confirm your SAS results, that there are about 9 million eligible beneficiaries. Also, common sense checks are very important. Look at your data, ask yourself if your results make sense.

MDR PROGRAMS

Programming Examples

Appendix A: Comparison of MDR Data files and M2 Data Files

M2 Table	Corresponding MDR File	Additional Filters/Notes
Eligibility/Population Summary	/mdr/pub/deers/summary/vm6agg/fy<fy>/fm<fm>/popagg.sas7bdat	Monthly SAS data sets.
Eligibility/<FY> DEERS Person Detail	/mdr/pub/deers/detail/vm6ben/fy<fy>/fm<fm>.>.txt	Limit to records where Primary Record Flag=1 and MHS Eligibility Flag=1. Monthly text files. All months but the most recent are compressed. Need to use special UNIX commands when reading in compressed files.
Relationships/<FY> Relationship Detail	/mdr/pub/deers/enr/vm6enr/fy<fy>/fm<fm>.sas7bdat	Monthly SAS data sets.
Health Care Services/Case Management	/mdr/pub/casemgmt/cm.sas7bdat	
Health Care Services/Direct Care/Ancillary Services/Radiology Detail	/mdr/pub/ancillary/fy<fy>/ancillary.sas7bdat	Use Record Type to separately identify Radiology records.
Health Care Services/Direct Care/Ancillary Services/<FY> Lab Detail	/mdr/pub/ancillary/fy<fy>/ancillary.sas7bdat	Use Record Type to separately identify lab records.
Health Care Services/Direct Care/Inpatient Admissions	/mdr/pub/sidr/fy<fy>/sidr.fy<fy>/fy<fy>.sas7bdat	Only contains raw records. If desired, merge in completion factor file to complete the data (/mdr/pub/sidr/compfac/fy<fy>/sidr.compfac.fy<fy>.txt.Z)
Health Care Services/Direct Care/<FY> Professional Encounters	/mdr/pub/sadr/fy<fy>.sas7bdat	
Health Care Services/Purchased Care/Institutional	/mdr/pub/tedi/fy<fy>/header.sas7bdat	
Health Care Services/Purchased Care/Non-Institutional/Non-Institutional, DHP	/mdr/pub/tedni/fy<fy>/champus.sas7bdat	Dual eligibles are in CHAMPUS and TDEFIC files. Need to remove from ONE of them.
Health Care Services/Purchased Care/Non-Institutional/Non-Institutional, MERHCF	/mdr/pub/tedni/fy<fy>/tdefic.sas7bdat	Dual eligibles are in CHAMPUS and TDEFIC files. Need to remove from ONE of them.
Health Care Services/Pharmacy/PDTS Summary	/mdr/pub/pdts/summary/fy<fy>/pdts.summary.fy<fy>/sum.sas7bdat	

M2 Table	Corresponding MDR File	Additional Filters/Notes
Health Care Services/Pharmacy/PDTS <FY>	/mdr/pub/pdts/detail/fy<fy>/pdts.detail.fy<fy>.txt.Z	
Health Care Services/Referrals	/mdr/pub/referral/referral.sas7bdat	
System Production/MEPRS	/mdr/pub/eas4/fy<fy>/eas4.fy<fy>/fy<fy>.sas7bdat	
System Production/MEPRS Personnel Detail	/mdr/pub/eas4/personnel/fy<fy>/eas4.personnel.fy<fy>/fy<fy>.sas7bdat	
System Production/WWR	/mdr/pub/wwr/fy<fy>/wwr.fy<fy>/fy<fy>.sas7bdat	
Reference Tables/APC Codes		
Reference Tables/APG Codes	/mdr/ref/apgref.txt	
Reference Tables/CPT/HCPCS Codes	/mdr/ref/cptref.cy<cy>.txt	
Reference Tables/DMIS ID Table	/mdr/ref/dmisid.index.fy<fy>.txt	This is a format file based on DMISID. Does not contain facility names.
Reference Tables/DRG Codes	/mdr/ref/drgref.fy<fy>.txt	
Reference Tables/ICD-9-CM Diagnosis Codes	/mdr/ref/icd9dxref.fy<fy>.txt	
Reference Tables/ICD-9-CM Procedure Codes	/mdr/ref/icd9procref.fy<fy>.txt	
Reference Tables/Market Area Table	/mdr/ref/cad.omni/a<cycm>.sas7bdat	
Reference Tables/MEPRS3 Codes	/mdr/ref/eas4.mepr3.fy<fy>/fy<fy>.sas7bdat	
Reference Tables/MEPRS4 Codes	/mdr/ref/eas4.mepr4.fy<fy>/fy<fy>.sas7bdat	
Reference Tables/Occupation Codes	/mdr/ref/eas4.occ.fy<fy>/fy<fy>.sas7bdat	
Reference Tables/Pharmacy ID(NCPDP)	/mdr/ref/pdts.ncpdp.fmt	This is a format file based on NCPDP ID. Need to determine what is in format file
Reference Tables/Purchased Care Providers	/mdr/pub/tedpr/tedpr.sas7bdat	
Reference Tables/Skill Type Codes	/mdr/ref/eas4.skill.fy<fy>/fy<fy>.sas7bdat	

Appendix C: Sample Programs

Example 1 – Create a format file

Objective: Create a format file containing a list of edi_pns that we identified w/ CHF.

```
libname in1 '/hpae2/kennell/linda';
filename out1 "/hpae2/kennell/linda/chf_edipn.fmt";

Data _null_;
  file out1;
  set in1.tempdata(keep=edi_pn) end=last;
  if _n_=1 then put 'PROC FORMAT; '/' VALUE $chf_pid';
  if not last then goto keep_it;
  if last then put @10 "" edi_pn $char10. ""=" ""T" /
    @10 ' other = "F";' /
    @1 'run;';
  delete;

  keep_it: put @10 "" edi_pn $char10. ""=" ""T";
  return;
run;
```

Example 2 – Apply DMISID format file

Objective: Apply DMISID format file to get enrollment service, region and HSSC region.

```
%include "/mdr/ref/dmisid.index.fy08.txt"; /* name of format is $par08X. */

enrsvc=substr(put(enrmtf,$par08X.),39,1); /* Branch of service */
enrreg=substr(put(enrmtf,$par08X.),31,2); /* Region – 01, 02, ... */
enrhreg=substr(put(enrmtf,$par08X.),40,1); /* HSSC region – N, S, W, O */
```

Example 3a – VM6 Example

Objective: Read in 1 month of DEERS VM6 file. Using a conditional read, keep primary records and MHS eligibles. Retain bencat and ACV. Count the number of TRICARE eligibles by bencat and ACV.

```
*
* Read in uncompressed VM6 file.
*;

filename invm6 "/mdr/pub/deers/detail/vm6ben/fy09/fm04.txt";

*
* Internal format statement that will be used to group prime vs non-prime ACV.
*;

proc format;
  values $primefmt
  'A', 'B', 'D', 'E', 'F', 'H', 'J', 'M', 'Q' = 'T'
  Other='F';
Run;

*
* Conditional read statement tells SAS to read mhselig and primary first. If the conditions
* (mhselig='1' and primary='1') are met then continue to read bencat and acv.
* Conditional read usually saves processing time.
* Group ACV into prime vs non-prime group using format statement.
*;

data deers(keep=bencat acv);
  infile invm6 missover lrecl=700;
  input @297 mhselig $char1.
    @307 primary $char1. @;
```

```

if (mhselig='1' and primary='1') then do;
    input  @390 bencat $char3.
           @539 acv   $char4.;
end;
else delete;

length prime $1;
if put(acv,$primefmt.)='T' then prime='Y';
   else prime='N';
run;

*
* Perform frequency procedure to count number of benes by bencat, acv and bencat*acv.
* The missing option gives you counts even if bencat or acv is blank.
* The list option display the frequency in a list format, instead of a table format.
*;

proc freq data=deers;
    tables prime*acv bencat acv bencat*acv/missing list;
    title 'FY09 FM04, Number of TRICARE eligibles by bencat and ACV';
run;

```

Example 3b – VM6 example using Macros

Objective: Read 4 months of DEERS VM6 file. Using conditional read keep primary records and MHS eligibles living in CONUS regions (01-12, AK). Retain common bencat, patient age and ACV. Derive beneficiary category group. Count number of TRICARE eligibles by derived beneficiary category group and ACV.

```

%macro readvm6(fy, fm, infile);

*
* Read in compressed VM6. Syntax is different than uncompressed file.
*;

filename invm6 pipe "uncompress -c /mdr/pub/deers/detail/vm6ben/fy&fy/fm&fm..txt";

*
* Conditional read statement tells SAS to read mhselig, primary and hssc_reg first.
* If the conditions (mhselig='1' and primary='1' and hssc_reg in N,S,W,A) are met then continue
to read rest of variables.
*;

data deers&fy&fm(keep=edi_pn bcat_grp acv);
    infile invm6 missover lrecl=900;
    input  @297 mhselig   $char1.
           @307 primary   $char1.
           @561 hssc_reg   $char1. @;
    if mhselig='1' and primary='1' then do;

        input  @287 patage 3.
               @309 comben $char1.
               @495 edi_pn $char10.
               @539 acv   $char1.;

    end;
else delete;

*
* Create bcat_grp variable using if-then-else statement.
*;

length bcat_grp $8;
if comben='4' then bcat_grp='AD';
   else if comben='1' then bcat_grp='ADD';

```

```

                else if comben in ('2' '3') and patage lt 65 then bcat_grp='NADD<65';
                else bcat_grp='NADD 65+';
run;

*
*   For each VM6 month, generate a cross tabulation of bcat_grp and acv.
*
*
proc freq data=deers&fy&fm;
    tables bcat_grp*acv/missing list;
    title "TRICARE eligibles, FY&fy. FM&fm.";
run;

%mend readvm6;

*
*   Call the macro READVM6 4 times – 1st call for FM 03, 2nd call for FM 06, etc.
*
*
%readvm6(fy=08, fm=03)
%readvm6(fy=08, fm=06)
%readvm6(fy=08, fm=09)
%readvm6(fy=08, fm=12)

```

Example 4a – SIDR Example

Objective: Read in SIDR. Keep dispositions that are delivery DRGs. Summarize number of dispositions, beddays, RWPs and fullcost by MTF, FY and FM.

```

libname insidr '/mdr/pub/sidr/fy08/sidr.fy08';

*
*   Conditionally read FY08 SIDRs records where DRG is 370-375. All other records
*   not meeting the condition will not be read.
*
*
data sidr;
    set insidr.fy08(keep=dmsdays totwrp fullcost mtf drg);
    where ('370' le drg le '375');
run;

*
*   Include QC checks using frequency to ensure code is doing what you are expecting.
*
*
proc freq data=sidr;
    tables drg/missing list;
    title 'QC: hould only include DRGs 370-375.';
run;

*
*   Summarize dmsdays, rwp and fullcost by mtf and drg.
*
*
proc summary data=sidr nway;
    class mtf drg/missing;
    var dmsdays totwrp fullcost;
    output out=outsum n=disp sum=dmsdays totwrp fullcost;
run;

*
*   Print out to SAS list the output from the summary.
*   Using VAR will output only the variables that you have listed. Excluding VAR will
*   output all variables in a file.
*
*

```

```

proc print data=outsum;
    var mtf drg disp dmisdays totwrp fullcost;
    title 'Final output, FY08 SIDR utilization by MTF and delivery DRGs.';
run;

```

Example 4b – SIDR Example

Objective: Read in SIDR. Get a list of EDI_PN who was ever diagnosed with asthma, in any DX position.

```

libname insidr '/mdr/pub/sidr/fy07/sidr.fy07';

*
*   Read in FY07 SIDR keeping all records. The array loops through each DX code to check
*   for asthma DX.

data sidr(keep=patuniq dx1-dx20 asthma);
    set insidr.fy07(keep=patuniq dx1-dx20);

    array xdx(20) $ dx1-dx20;
    length asthma $1;
    asthma='N';

    do i=1 to 20;
        if substr(left(xdx(i)),1,3)='493' then asthma='Y';
    end;

    if asthma='Y';
run;

*
*   QC code above using frequency and print procedures.
*;

proc freq data=sidr;
    tables asthma/missing list;
    title 'QC: all records should equal Y';
run;

proc print data=sidr(obs=50);
    var dx1-dx10 asthma;
    title 'QC: sample print out of records';
run;

*
*   Sort nodupkey removes duplicate BY variable, in this case removes duplicate EDIPN.
*   Print these EDIPNs.
*;

proc sort data=sidr(keep=patuniq) nodupkey;
    by patuniq;
run;

proc print data=sidr;
    var patuniq;
    title 'List of EDI_PNs with asthma DX in FY07';
run;

```

Example 5a – SADR Example

Objective: Read in 1 year of SADRs. Count the number of mental health encounters and sum historical and raw work RVU by year-month date and sponsor service.

```

libname insadr '/mdr/pub/sadr;

```

```

*
* Read in FY08 SADR using conditional read where mdc=19 (mental health).
* You could use either the "where" or "if" statement. "Where" statement only read in records that
* meet the criteria. Keep ambulatory meprs codes.
* Variable encounter is created and has value for 1 for each record.
* Substring encounter date keeping year and month.
*
*

```

```

data mh_sadr;
  set insadr.fy08(keep=mdc meprscd sponsvc rvuhist rrvu encdate);
  where (mdc='19');
  if substr(meprscd,1,1)='B' or substr(meprscd,1,3)='FBI';

  length encounter 3;
  encounter=1;
  encdate_yrnm=substr(encdate,1,6);
run;

```

```

*
* Generate number of records (n) and summarize (sum) statistics for variables encounter,
* adjusted RVU, historical RVU and work RVU.
*
*

```

```

proc means data=mh_sadr sum;
  class encdate_yrnm sponsvc;
  var encounter rvuhist rrvu;
  title 'Number of MH encounters and RVUs by date & sponsor service, FY08';
run;

```

Example 5b – SADR Example

Objective: Read in 2 years of SADR. Append each yearly SADR files to 1 cumulative file. Compute the number of PTSD encounters by FY, FM and MTF.

```

%macro readsadr(fy);

  Libname insadr "/mdr/pub/sadr";

  *
  * Macro that takes 1 SADR year at a time and read in encounters with
  * ambulatory meprs code. Convert encounter date from character to SAS date
  * which makes it easier to manipulate.
  * Loop through 4 DX codes and flag the record if any DX code has PDTS diagnosis.
  * Keep only encounters with PDTS DX in any position.
  *
  *
  data sadr&fy.(keep=dmsid edate);
    set insadr.fy&fy.(keep=meprscd dmsid encdate icd1-icd4);
    where (substr(meprscd,1,1)='B' or substr(meprscd,1,3)='FBI');
    edate=input(encdate,yymmdd8.);

    array xdx(4)$ icd1-icd4;

    length ptsd $1;
    ptsd='N';

    do i=1 to 4;
      if substr(left(xdx(i)),1,5)='30981' then ptsd='Y';
    end;

    if ptsd='Y';
run;

*

```

```

*       Appends or add to the end of the file each quarterly SADR to a file called
*       allsadr.
*
proc append data=sadr&fy. base=allsadr;
run;

*
*       Once the quarterly file is appended delete from temp work space.
*
proc datasets nolist;
       delete sadr&fy;
run;

%mend readsadr;

*
*       Calls the macro passing each FY and quarter each time.
*
%readsadr(fy=08)
%readsadr(fy=07)

*
*       Generate crosstab tables for encounter date and dmsid. Since encounter date (edate)
*       is a SAS date then we can ask the output in MMMYY format (eg. OCT08, NOV08, etc.).
*
proc freq data=allsadr;
       Tables edate*dmsid/missing list;
       Format edate monyy5.;
       Title 'Final output, number of PTSD encounters by FY, FM, & MTF';
Run;

```

Example 6a – TEDI Example

Objective: Read in 2 years of TEDI. Count number of admissions and gov't paid for prime enrollees to MTF 0124 by bencat.

```

libname intedi07 '/mdr/pub/tedi/fy07';
libname intedi08 '/mdr/pub/tedi/fy08';
libname out1 '/hpae2/kennell/linda';

*
*       Conditional read of TEDI FY07 and FY08 keeping records for prime enrollees to MTF0124.
*
data tedi;
       set intedi07.header(keep=fy acv denrsite adm paid bencat)
           intedi08.header(keep=fy acv denrsite adm paid bencat);
       where (acv in ('A', 'B', 'D', 'E', 'F', 'H', 'J', 'M', 'Q') and denrsite='0124');
run;

*
*       Summarize the number of admissions and gov't amt paid by FY and bencat.
*       Since we are using "class fy bencat" then we do not have to sort before doing summary.
*
proc summary data=tedi nway;
       class fy bencat/missing;
       var adm paid;
       output out=sumtedi sum=;
run;

```

```

*
*      Output results from the summary to a SAS dataset. Perform proc contents to see number of
*      records and list of variables and their attributes.
*.;

data out1.prime0124_fy07fy08;
    set sumtedi(keep=fy bencat adm paid);
run;

proc contents data= out1.prime0124_fy07fy08;
run;

```

Example 6b – TEDI Example

Objective: Read in 1 year of TEDI. Excluding dual eligibles and TFL claims, compute the top 20 DRGs based on gov't paid.

```

libname intedi08 "/mdr/pub/tedi/fy08";
filename outdrg "/hpa2/kennell/linda/top20drgs.txt";

```

```

*
*      Read in FY08 TEDI file and keep records where MERCHF flag is A or N. "If" statement will
*      read in all records then check for tflflag condition – this is usually less efficient
*      then using "where" statement.
*.;

```

```

data tedi;
    set intedi08.header(keep=tflflag drg adm paid);
    if tflflag in ('A' 'N');
run;

```

```

*
*      Since the summary procedure uses "by drg" then we have to sort by drg first.
*.;

```

```

proc sort data=tedi; by drg;
run;

```

```

*
*      Summarize the number of admissions and govt amt paid by drg.
*.;

```

```

proc summary data=tedi nway;
    by drg;
    var adm paid;
    output out=sumtedi sum=;
run;

```

```

*
*      Perform Univariate statistics of admission and paid by DRG.
*.;

```

```

proc univariate data=sumtedi;
    class drg;
    var adm paid;
run;

```

```

*
*      Sort the output from summary by descending govt amt paid.
*.;

```

```

proc sort data=sumtedi(keep=drg adm paid); by descending paid;
run;

```

```

*
```

```
*      Print out top 20 DRGs based on govt amt paid. Can print out all DRGs by removing obs=20.
*;
```

```
proc print data=sumtedi(obs=20);
  var drg adm paid;
  title 'Final output, top 20 DRGs by govt amount paid';
run;
```

```
*
*      Output the all the DRGs and govt amt paid to a ASCII fixed length delimited (!) file.
*;
```

```
data _null_;
  set sumtedi;
  file outdrg lrecl=22;
  retain delim '!';
  put @1 drg      $char3.          delim $char1.
      adm      5.                delim $char1.
      paid    10.2              delim $char1.;
run;
```

Example 7a – TEDni example

Objective: Read in 1 year of TEDni, excluding RX claims. Keep claims from a list of continuous prime enrollees (from format file). Count number of mental health (based on MDC) visits and gov't paid by FY, FM, and bencat.

```
libname tedn "/mdr/pub/tedni/fy08";
filename outmh "/hpa2/kennell/linda/tedni_fy08_mh.txt";
```

```
*
*      Load format file into the program. This format contains list of continuous
*      prime enrollees.
*;
```

```
%include "/hpa2/kennell/continuous_prime.fmt"; /* Format name is $edipnfmt */
```

```
*
*      Read in FY08 DHP file keeping only claims for the list of people in the format file.
*      This greatly reduced processing time.
*      Removed pharmacy records. Retain records with a mental health MDC.
*;
```

```
data tedn(keep=fy fm bencat visits paid);
  set tedn.champus(keep=edi_pn bencat fy fm pic cpt taxid mdc visits paid);
  where (put(edi_pn,$edipnfmt.)='T');
  if taxid='431867735' or pic='D' or cpt='98800' then delete;
  if mdc='19';
```

```
run;
```

```
*
*      Summarize number of visits and amount paid by fy, fm, and bencat. Do not have to sort
*      fy, fm, bencat because we are using a CLASS instead of BY statement – MORE EFFICIENT!
*;
```

```
proc summary data=tedn nway;
  class fy fm bencat/missing;
  var visits paid;
  output out=outsum sum=;
```

```
run;
```

```
*
*      Output summary statistics to an ASCII variable length delimited (!) file.
*;
```

```

data _null_;
    set outsum;
    file outmh delimiter="!";
    put @1 fy fm bencat visits paid;
run;

```

Example 7b – TEDni example

Objective: Read in 1 year of TEDni, excluding RX claims. Compute total govt paid and OHI by HSSC residence region, 3-digit patzip and bencat group.

```

libname tedn "/mdr/pub/tedni/fy08";
filename outtedni "/hpae2/kennell/linda/tedni_fy08_paidohi.txt";

*
*   Remove Medicare<65(tfiflag=U) since these claims are in both TEDni files.
*   Remove RX claims.
*   Create bcat variable. Substring patzip to first 3 digits.
*
data tedn;
    set tedn.champus(keep=resreg patzip pic cpt taxid paid comben patage tfiflag ohi in=dhpfile)
        tedn.tdefic(keep=resreg patzip pic cpt taxid paid comben patage tfiflag ohi in=tfifile);
    if tfifile and tfiflag='U' then delete;
    if taxid='431867735' or pic='D' or cpt='98800' then delete;

    length bcat $7 patzip3 $3;
    if comben='4' then bcat='AD';
        else if comben='1' then bcat='ADD';
        else if patage lt 65 then bcat='NADD<65';
        else bcat='NADD65+';
    patzip3=substr(patzip,1,3);
run;

*
*   Summarize govt paid and OHI by region, patzip3 and bencat. Option missing generates
*   statistics if any of the class variable is missing.
*
proc summary data=tedn nway;
    class resreg patzip3 bcat/missing;
    var paid ohi;
    output out=outsum sum=;
run;

*
*   Output summarized file in ASCII fixed length file. Max default of lrecl is 80 bytes.
*   If your output file is longer than 80 then need to specify record length otherwise
*   file will wrap.
*
data _null_;
    set outsum;
    file outtedni lrecl=35;
    put      @1 resreg      $char2.
            @3 patzip3    $char3.
            @6 bcat       $char7.
            @13 paid      12.2
            @25 ohi       10.2;
run;

```

Example 8a – PDTS Example

Objective: Read in 1 year of PDTS, excluding clinician administered scripts. For TRICARE prime enrollees living in the US, compute the number scripts, days supply and gov't costs by residence region and beneficiary category. Output to SAS dataset.

```
filename inpdts pipe "uncompress -c /mdr/pub/pdts/detail/fy08/pdts.detail.fy08.txt.Z";
libname idtable "/mdr/ref/dmisid.index/";
libname out1 '/hpae2/kennell/linda';
```

```
*
*       Read in DMIS Table to get MTF dispensing fee for fullcost calculation.
*;
```

```
proc sort data=idtable.fy08(keep=dmisid rxoth rxmilpy) out=idtable; by dmisid;
run;
```

```
data all;
  infile inpdts lrecl=700 missover;
  input  @313 source  $char1. @;
  if source in ('D' 'T' 'M') then do;
    input
      @40 dayssup  3.
      @43 icost    10.2
      @53 sdfee    6.2
      @75 copay    7.2
      @318 amtpaid 10.2
      @340 region  $char2.
      @348 comben  $char1.
      @353 acv     $char1.
      @363 dmisid  $char4.
      @367 patage  3.
    ;
  end;
  else delete;

  if acv in ('A' 'B' 'D' 'E' 'F' 'H' 'J' 'M' 'Q');
  if region in ('01' '02' '03' '04' '05' '06' '07' '08' '09' '10' '11' '12' 'AK');

  length bencat $7;
  if comben='4' then bencat='AD';
  else if comben='1' then bencat='ADD';
  else if patage lt 65 then bencat='NADD<65';
  else bencat='NADD65+';
```

```
run;
```

```
*
*       Perform QC.
*;
```

```
proc freq data=all;
  tables acv region source bencat*comben/missing list;
  title 'QC: frequency on filtered & derived variables';
run;
```

```
proc means data=all n min max;
  class bencat;
  var patage;
  title 'QC: check patage used in bencat derivation';
run;
```

```
*
*       Merge DMISID table to get MTF dispensing fee. Derive fullcost.
*;
```

```
proc sort data=all; by dmisid;
run;
```

```

data all(keep=region bencat fullcost dayssup script);
  merge all(in=master) idtable; by dmsid;
  if master;

  if source in ('C' 'D') then dispfee=rxoth + rxmilpy;
  else dispfee=sdfree;

  if source in ('T' 'M') then fullcost=amtpaid+copay;
  else if source in ('C' 'D') then fullcost=dispfee+icost;

  script=1;
run;

proc summary data=all nway;
  class region bencat/missing;
  var script dayssup fullcost;
  output out=outsum sum=;
run;

data out1.pdts_prime_util(keep=region bencat script dayssup fullcost);
  set outsum;
run;

```

Example 8b – PDTS example

Objective: Read in 2 years of PDTS, excluding clinician administered scripts. For each year count the number of scripts by therapeutic class. Merge number of scripts from both years together into 1 file and output results.

```

filename pdts07 pipe "uncompress -c /mdr/pub/pdts/detail/fy07/pdts.detail.fy07.txt.Z";
filename pdts08 pipe "uncompress -c /mdr/pub/pdts/detail/fy08/pdts.detail.fy08.txt.Z";

filename outpdts "/hpa2/kennell/linda/pdts_theraclass_fy07fy08.txt";

*
*   Read in FY07 and FY08 PDTS files keeping MTF, MCSC and TMOP scripts.
*;

data pdts07(keep=theraclass script07);
  infile pdts07 lrecl=700 missover;

  input @313 source      $char1. @;
  if source in ('D' 'M' 'T') then do;
    input @259 theraclass $char6.;
  end;
  else delete;

  script07=1;
run;

data pdts08(keep=theraclass script08);
  infile pdts08 lrecl=700 missover;

  input @313 source      $char1. @;
  if source in ('D' 'M' 'T') then do;
    input @259 theraclass $char6.;
  end;
  else delete;

  script08=1;
run;

*
*   Summarize each file by therapeutic class to get total number of scripts by therapeutic class.
*;

```

```

proc summary data=pdts07 nway;
  class theraclass/missing;
  var script07;
  output out=sum07 sum=;
run;

proc summary data=pdts08 nway;
  class theraclass/missing;
  var script08;
  output out=sum08 sum=;
run;

*
*   Since the PDTS files are quite large, remove these files after summary will free up temporary
*   work space.
*
proc datasets nolist;
  delete pdts07 pdts08;
run;

*
*   Sort each summarized file by therapeutic class so that we can merge the 2 files together.
*   Output merged file to an ascii delimited file.
*
proc sort data=sum07(keep=theraclass script07); by theraclass;
run;

proc sort data=sum08(keep=theraclass script08); by theraclass;
run;

data final;
  merge sum07 sum08; by theraclass;
  if script07=. then script07=0;
  if script08=. then script08=0;
run;

data _null_;
  set final;
  file outpdts lrecl=22;
  retain delim '!';
  put @1  theraclass      $char6.          delim $char1.
        script07 10.      delim $char1.
        Script08      10          delim $char1.;
run;

```

Example 9a – multiple data types example

Objective: For the purpose of disease management, generate a list of patient IDs meeting the diabetes definition below. Note, diabetes definition below is used to demonstrate capability of SAS programming and is NOT recommended for any DM analysis.

Diabetes DM criteria (patient must meet one of the following within 12 months):

- 5 or more outpatient visits with diabetes DX in any DX position (DX=250)
- 1 or more inpatient admission with diabetes DX in any DX position (DX=250) or diabetes DRG (DRG=294,295)
- 1 or more ER visit (CPT 99281-99285 or place of svc=23) with diabetes DX in any DX position (DX=250)
- 4 or more diabetic scripts

```

libname insidr '/mdr/pub/sidr/fy08/sidr.fy08';
libname insadr '/mdr/pub/sadr';

```

```

libname intedi '/mdr/pub/tedi/fy08';
libname intedni '/mdr/pub/tedni/fy08';
filename inpdts pipe "uncompress -c /mdr/pub/pdts/detail/fy08/pdts.detail.fy08.txt.Z";

filename outdiab "/hpa2/kennell/linda/diab_edipn.txt";

%include "/hpa2/kennell/linda/diabetes_ndc.fmt"; /* contains list of diabetes NDCs */

*
*   Read in SIDR to get number of DC diabetes admission.
*
data sidr(keep=patuniq sidr_diab rename=patuniq=edi_pn);
  set insidr.fy08(keep=patuniq dx1-dx20 drg);

  array xdx(20) $ dx1-dx20;
  length diabetes $1;
  diabetes='N';

  do i=1 to 20;
    if substr(left(xdx(i)),1,3)='250' then diabetes='Y';
  end;

  if drg in ('294' '295') then diabetes='Y';
  if diabetes='Y';
  sidr_diab=1;
run;

*
*   Read in SADR to get number of DC diabetes encounters.
*
data sadr(keep=patuniq sadr_diab sadr_diab_er rename=patuniq=edi_pn);
  set insadr.fy08(keep=patuniq icd1-icd4 meprscd);
  if substr(meprscd,1,1)='B' or substr(meprscd,1,3)='FBI';

  array xdx(4) $ icd1-icd4;
  length diabetes $1;
  diabetes='N';

  do i=1 to 4;
    if substr(left(xdx(i)),1,3)='250' then diabetes='Y';
  end;

  if diabetes='Y';
  sadr_diab=0;
  sadr_diab_er=0;

  if substr(meprscd,1,2)='BI' then sadr_diab_er=1;
  else sadr_diab=1;
run;

*
*   Read in TEDI to get number of PC diabetes admission.
*
data tedi(keep=edi_pn tedi_diab);
  set intedi.header(keep=edi_pn dx1-dx12 drg);

  array xdx(12) $ dx1-dx12;
  length diabetes $1;
  diabetes='N';

  do i=1 to 12;
    if substr(left(xdx(i)),1,3)='250' then diabetes='Y';
  end;

```

```

        if drg in ('294' '295') then diabetes='Y';
        if diabetes='Y';
        tedi_diab=1;
run;

*
*   Read in TEDni to get number of PC diabetes visits.
*;

data tedni(keep=edi_pn tedni_diab tedni_diab_er);
    set intedni.champus(keep=edi_pn dx1-dx8 cpt place tflflag in=dhp)
        intedni.tdefic(keep=edi_pn dx1-dx8 cpt place tflflag in=tfli);
    if tflflag='U' and tfl then delete;

    array xdx(8) $ dx1-dx8;
    length diabetes $1;
    diabetes='N';

    do i=1 to 8;
        if substr(left(xdx(i)),1,3)='250' then diabetes='Y';
    end;

    if diabetes='Y';
    tedni_diab=0;
    tedni_diab_er=0;

    if cpt in ('99281' '99282' '99283' '99284' '99285') or place='23' then tedni_diab_er=1;
    else tedni_diab=1;
run;

*
*   Read in PDTS to get number of diabetes scripts.
*;

data pdts(keep=edi_pn pdts_diab);
    infile inpdts lrecl=700 missover;
    input @141 ndc $char11.
        @313 source $char1. @;
    if source in ('D' 'M' 'T') and put(ndc,$diabndc.)='T' then do;
        input @392 edi_pn $char10.;
    end;
    else delete;

    pdts_diab=1;
run;

*
*   Append SIDR, SADR, TEDi, TEDni, and PDTS data sources together and calculate the number of
admissions, visits, and scripts.
*;

data all(keep=edi_pn inpat_diab outpat_diab er_diab script_diab);
    set sidr(keep=edi_pn sidr_diab)
        sadr(keep=edi_pn sadr_diab sadr_diab_er)
        tedi(keep=edi_pn tedi_diab)
        tedni(keep=edi_pn tedni_diab tedni_diab_er)
        pdts(keep=edi_pn pdts_diab);

    if sidr_diab=. then sidr_diab=0;
    if sadr_diab=. then sadr_diab=0;
    if sadr_diab_er=. then sadr_diab_er=0;
    if tedi_diab=. then tedi_diab=0;
    if tedni_diab=. then tedni_diab=0;
    if tedni_diab_er=. then tedni_diab_er=0;

```

```

if pdts_diab=. then pdts_diab=0;

inpat_diab=sidr_diab+tedi_diab;
outpat_diab=sadr_diab+tedni_diab;
er_diab= sadr_diab_er+tedni_diab_er;
script_diab=pdts_diab;
run;

*
*   Remove temporary datasets that are no longer needed.
*
proc datasets nolist;
    delete sidr sadr tedi tedni pdts;
run;

*
*   Summarize number of admissions, visits and scripts by person id.
*
proc summary data=all nway;
    class edi_pn/missing;
    var inpat_diab outpat_diab er_diab script_diab;
    output out=outsum sum=;
run;

*
*   Keep people who met diabetes criteria.
*
data outsum;
    set outsum;
    if substr(edi_pn,1,1)=' ' then delete;
    if (outpat_diab ge 5) or (inpat_diab ge 1) or (er_diab ge 1) or (script_diab ge 4);
run;

*
*   Output list of people to text file.
*
data _null_;
    set outsum;
    file outdiab;
    retain delim '!';
    put @1 edi_pn          $char10.          delim $char1.
        inpat_diab       3.                  delim $char1.
        outpat_diab      5.                  delim $char1.
        er_diab          5.                  delim $char1.
        script_diab      5.                  delim $char1. ;
run;

```