MDR Functional User's Handbook

Version 2.5
Prepared for: Office of the Assistant Secretary of Defense (Health Affairs) / Defense Health Agency

Prepared by: Kennell and Associates

Updated: 12/19/2016

Table of Contents

List of Tables

Purpose of this Document

This document is prepared for new functional users of the MHS Data Repository (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.

When the MDR moves to Capacity Services, SAS Enterprise Guide (EG) will be available to all users. This software package is considered more user-friendly than UNIX and Base SAS and includes a query builder with more point and click capability. However, basic knowledge of SAS code is still necessary even when using EG.

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, such as the MHS Mart (M2).

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 access, contact the Help Desk by phone at 1-800-600-9332 or by email at DHA.SDDAccess@mail.mil. If you cannot connect to the OOB via VPN, you may need to contact the Defense Information Systems Agency (DISA) Montgomery at 334-416-3472.

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 and Associates) to discuss important differences that may affect your results.

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 VM6 files have evolved from DEERS Point in Time Extract (PITE) and VM4 files. 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 since9/11/2001. 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.

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.

The major CHCS extracts in the MDR include:

Table 1: The Major CHCS/AHLTA Extracts

Extract	Nickname	Record Definition	Sent	Processed
Ancillary		Outpatient (and some inpatient) Lab/Rad/Rx[1]	Monthly	Monthly
Appointment		Appointment	Weekly	Weekly

[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. Ancillary records from AHLTA are also in the CDR files in the MDR.

Extract	Nickname	Record Definition	Sent	Processed
Clinical Data Repository	CDR	Includes Appointment, Lab and Rad Results, Immunizations, Medications, Patient, and Vitals AHLTA files	Weekly	Weekly
Comprehensive Professional Record	CAPER	Enhanced SADR	Weekly	Weekly
Schedulable Entity		Lists schedulable appointment slots from CHCS	Daily	Weekly
Standard Ambulatory Data Record	SADR	Professional Outpatient + Rounds	No Longer Updated: Data spans FY98-FY12	
Standard Inpatient Data Record	SIDR	Inpatient Hosp Record	Monthly	Monthly

For each of these files, the MDR contains enhanced data files that include popular data elements such as enhanced and provider aggregate RVUs, MS-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.

CDR

The MDR contains clinical data from AHLTA that had previously been housed in the defunct Clinical Data Mart (CDM). This data includes vitals, all radiology orders and results, all laboratory orders and results, medication orders and fills, and immunizations. This data is considered more complete than the ancillary data in the MDR because it contains all orders for radiological tests, laboratory tests, and prescriptions, even if some orders have not been completed. This data also includes care that occurred in an inpatient setting. The CDR data for the current FY is processed weekly.

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, TMOP, VA, and other locations.

Referrals

Referral data from the CHCS hosts is transmitted to the MDR weekly. Referral data could be linked to the initial appointment that generated the referral and any appointment made with a specialty provider or clinic for that referral within the MTF. TRICARE regions have worked with the MDR team to devise

linkages called the Unit Identifying Number (UIN) from MTF-generated referrals to appointments made against those referrals in the network. This allows for much more complete referral pattern analysis.

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.

Purchased care data began coming into the MDR as Health Care Service Records (HCSR) and now all claims are available on a daily basis in the TRICARE Encounter Data (TED) Operational Data Store (ODS). These claims are sent into the Provider Encounter Processing and Reporting (PEPR) system in Aurora. DHA 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.
- Dental Care: Claims and Provider Records for Active Duty and Retirees

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. Purchased care dental claims are also available in the MDR for TDP and TRDP.

Other Data Files

The MDR houses numerous other specialized data files. The Overseas Contingency Operation Injured, Ill, and Wounded (IIW) file contains health information about service members who were injured or killed serving in Operation Enduring Freedom (OEF) and Operation Iraqi Freedom (OIF). The IIW file is located in the MDR's pub directory. The Health Risk file is also housed in the pub directory and contains diagnosis information about patients and a health risk score based on a person's medical history that rates their care costliness relative to other patients in the MHS.

The MDR has other specialized files in its restricted directory. The Contingency Tracking System (CTS) is a restricted file that contains deployment information. The FHIE files are also restricted and contain details about individuals who have separated from the military can be found on the MDR in the restricted FHIE files. Theater medical data is also housed on the MDR in the restricted TMDS file. Any

restricted files on the MDR can be accessed upon submission and approval of an MHS Data Repository Special Justification form on which users must specify the reason they need access to the data and a valid Data Use Agreement (DUA) number. DHA's Decision Support Division (POC: James Huber-james.t.huber4.civ@mail.mil) initially accepts these forms and begins processing them for approval.

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 Defense Health Agency (DHA) Decision Support Division (DSD) (james.t.huber4.civ@mail.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
RESTRICTED	Special Justification	Sensitive MDR Analytical Data Sets
APROD	MDR Project Files; not generally accessible	
LOG		
INTERIM (INT)		
RESIDUAL		
RAW		
ARS	Special Justification	Data feeds for M2
APUB		Archives of public files
AREF		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. This is where a user's profile resides.

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. The DHA DSD 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
Appointment	/mdr/pub/appt/detail/fy<fy>/fy<fy>.sas7bdat
Case Management	/mdr/pub/casemgmt/cm.sas7bdat
CDR files	/mdr/pub/cdr/*/fy<fy>/
DEERS Beneficiary Level	/mdr/pub/deers/detail/vm6ben/fy<fy>/fm<fm>.sas7bdat (.txt prior to FY11)
DEERS Enrollment	/mdr/pub/deers/enr/vm6enr/fy<fy>/fm<fm>.sas7bdat
Health Risk	/mdr/pub/riskadjustment/fy<fy>/health_risk.sas7bdat
Injured, Ill, Wounded (IIW)	/mdr/pub/iiw/iiw.sas7bdat
MEPRS	/mdr/pub/eas4/fy<fy>/eas4.fy<fy>/fy<fy>.sas7bdat
MTF Ancillary Lab & Rad	/mdr/pub/ancillary/fy<fy>/ancillary.fy<fy>/fy<fy>.sas7bdat
MTF Inpatient	/mdr/pub/sidr/fy<fy>/sidr.fy<fy>/fy<fy>.sas7bdat
MTF Professional/CAPER	/mdr/pub/caper/fy<fy>.sas7bdat; /mdr/pub/caper/enhanced/fy<fy>.sas7bdat
Pharmacy	/mdr/pub/pdts/detail/fy<fy>/fy<fy>.sas7bdat
Referral	/mdr/pub/referral/referral.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

CDR * could be one of the following: procedures, med, micro, vitals, appt, chem, rad, path, patient, imm

Most of these data files were discussed in the "Overview of Data Types" section above. The MEPRS content in the table above comes from the Expense Assignment System (EAS), which is a local management accounting system 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 extracts 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.

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. Selected files in the MDR REF directory and useful reference files in the pub directory are:

Table 4: MDR Ref Directory

Content	MDR Directory Path and File Name
APG Weight Table	/mdr/ref/apgref.txt
Catchment Area Directory	/mdr/ref/cad.omni/a<cycm>.sas7bdat
CPT Weight Table	/mdr/ref/rvu.cy<cy>/rvumast.sas7bdat
DMISID Reference Table	/mdr/ref/dmisid.index.fy<fy>.txt
ICD-9 Diagnosis Descriptions	/mdr/ref/icd9dxref.fy<fy>.txt
ICD-9 Procedure Descriptions	/mdr/ref/icd9procref.fy<fy>.txt
ICD-10 Diagnosis Descriptions	/mdr/ref/icd10dxref.fy<fy>.txt
ICD-10 Procedure Descriptions	/mdr/ref/icd10procref.fy<fy>.txt
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
MS-DRG Weight Tables	/mdr/ref/msdrgref.fy<fy>.txt
NPPES	/mdr/pub/nppes/nppes.sas7bdat
Purchased Care Provider Table	/mdr/pub/tedpr/tedpr.sas7bdat

The NPPES file in the MDR provides National Provider IDs (NPIs), provider names, and DEERS Electronic Data Interchange Person Numbers (EDIPNs) if the provider works in the direct care system.

For M2 users, Appendix C 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 B for information about how MDR and M2 files compare.

Many electronic resources can be found at http://www.health.mil/Military-Health-Topics/Technology/Support-Areas/MDR-M2-ICD-Functional-References-and-Specification-Documents. This website is prepared for the functional proponent of the MDR, Defense Health Agency (DHA).

Table 5: MDR and M2 Resources

Type of File	Purpose
Interface Control Documents	Describes flow of data from sources to MDR; lists and defines raw data source fields
MDR Functional Specifications	Describes processing of data to prepare files in mdr/pub from the raw data sources and gives detailed definitions of data elements
MDR Data Dictionary	Describes content of each data element in each file in the MDR

M2 Functional Specifications	Describes M2 extract and linking
M2 Data Dictionary	Describes content of each data element in each file in the M2

The MDR Functional Specifications and Interface Control documents are critical for understanding the definitions of data elements beyond what is found in the MDR Data Dictionary. These documents and the MDR Data Dictionary are all critical resources for programmers and will be referenced 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

Functional Category	Command	Use	Example	Example Explanation
Navigating among, maintaining, and modifying directories and files	cd	Navigate among directories	cd /mdr/pub cd .. cd	Change directories to /mdr/pub Go back 1 directory Go to home directory
	pwd	Shows what directory you are in (where am I?)		
	ls	Shows contents of the working directory	ls -lrt	List the contents in the long form (with file permissions and size), and sort recursively by time modified
	mkdir	Creates a new directory	mkdir data	Creates a new directory called data in the current directory

Functional Category	Command	Use	Example	Example Explanation
	chmod	Change the permissions on a directory or file	chmod 770 test.sas	Changes the permissions so that the owner and all members of his/her group can read, write, and execute test.sas
	cp	Copy files or directories within or across directories	cp ../test.sas . cp oldfile newfile	Copies test.sas from the directory above the current directory into the current directory
	mv	Move or rename files or directories within or across directories	mv test.sas data/sample.sas mv oldfile newfile	Moves test.sas to the directory called data, and renames it as sample.sas
Determining and controlling disk space usage	df	Show free disk space	df -g . df -k .	Shows the disk space (in GB) of the current directory Shows amount of space the individual user is using
	gzip	Use gzip to compress files	gzip sample.txt	Compresses sample.txt
	gunzip	Use gunzip to uncompress files	gunzip sample.txt.gz	Uncompresses sample.txt
	ctrl+z	Suspend a running process		
Writing, submitting and monitoring SAS programs	pico	Open PICO text editor	pico test.sas	Opens up test.sas for editing
	sas	Submit a program in the /saswork area	sas test.sas	Submit test.sas
	sasbig	Submit a program in the /bigsaswork area	sasbig test.sas -memsize 1G	Submit test.sas with 1GB of memory
	llq	Lists which SAS programs are running by username and time		

Functional Category	Command	Use	Example	Example Explanation
	sasstop	Cancel a SAS program		Stops a program from running. Use 'llq' to get job_id. Only enter the numbers between the two periods of the job_id.
Help with UNIX commands	whatis	Describe what a command is	whatis cd	Briefly describe what the "cd" command does
	man	Shows the manual page for a UNIX command	man ls	Show the manual page for the "ls" command
Miscellaneous	alias	Define synonym or shortcut commands	alias home='cd /hpae2/kennell/keith'	Set up a short cut in user profile (see Appendix A) so that, when you type home, it takes you to that directory

Additional UNIX commands can be found in Appendix C.

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 MDR[2]. 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.

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

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.

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

```
drwxrwx---    2 tcomer   shken          256 May 16 09:52 formats
-rw-r-----    1 tcomer   shken         3642 May 23 06:12 Inpt_Test_HU.txt
drwxrwx---    2 tcomer   shken         4096 May 24 08:50 navypop
```

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 'formats' and 'navypop' are directories. The '-' in the row below indicates that Inpt_Test_HU.txt 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.

The ls –lrt command as well as other frequently used commands can also be aliased to a shorter form in an individual's .profile (see Appendix A).

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

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 Corporate and Service Node User's Guide for the command line used to submit SAS jobs when implementing the suggested "Keeping Logs" macro or Appendix E, Example 1 of this document.

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 or notepad and then uploaded to your directory via WinSCP. 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.

The PICO editor accessed through Putty 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
Exit current operation	[Ctrl] Z	
Search/Find text in the file	[Ctrl] W	
Go to beginning of the line	[Ctrl] A	
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 Programs on 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). Then, the computer runs the program and returns back a log of how the program behaved, which is 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 or 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 (Link: (http://support.sas.com/documentation/cdl/en/lrdict/64316/HTML/default/viewer.htm#a002237888.htm). 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:

- compress= YES or CHAR or BINARY: Compresses medium to large portions of repeated characters (YES, CHAR options) or numeric variables (BINARY option).

- 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). You can also set obs equal to 'max' if you do not want a limit.

- 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 in the log. Macros are a specific type of SAS utility, which will be addressed (lightly) later in this document.

- ls: Short form of linesize. Specifies the character length of each line for display and print purposes. The ls value can range from a minimum value of 64 to a maximum value of 256.

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 for FY16, the file name, /mdr/pub/sidr/fy16/fy16.sas7bdat shows. In the file name, the member is the fy16.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'. In this case, the libname would be input into a program as:

```
libname insidr '/mdr/pub/sidr/fy16';
```

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, which in this case is 'insidr'. The libref label must begin with a letter or underscore and must not contain any special characters (only letters, numbers and the underscore). For example, the reference to this SIDR member in a set statement would appear as:

```
set insidr.fy16;
```

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.cy16.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 deers10 pipe "uncompress -c /mdr/pub/deers/detail/vm6ben/fy10/fm12.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 eventually all compressed files will be removed from the MDR. In the meantime, this technique must be used.

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.

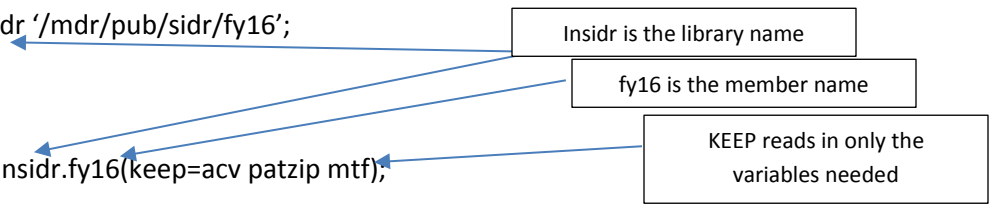
The following code will read in three data elements in the first 100 observations of the SAS dataset '/mdr/pub/sidr/fy16/fy16.sas7bdat'.

```
options nocenter obs=100;

libname insidr '/mdr/pub/sidr/fy16';

data sidr16;

    set insidr.fy16(keep=acv patzip mtf);
```



The ‘data’ statement creates the temporary data set called ‘sidr16’.
The ‘set’ statement that follows is used with SAS datasets only.

The ‘keep’ statement is very important. If there is no ‘keep’ statement, 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 and there should be one space between each variable name. The appendix of programs (Appendix D) 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. Many text data files on the MDR have been converted to SAS files but some DEERS VM6 files are still in text from FY10 back to FY98. MDR users may also encounter text files in certain reference files they want to use in their programs.

There are two types of text files: fixed length and delimited. The statement for reading in an uncompressed fixed length file is:

```
filename inpop “/mdr/pub/deers/detail/vm6ben/fy09/fm04.txt”;

data pop;
        infile inpop missover lrecl=700;

        input
                @297 mhselig $char1.
                @389 primary $char1.
                @403 bencat  $char3.
                @495 edi_pn $char10.
                @539 acv $char1.;
```

INPOP is the FILEREF

“$” = Character variable

The data step creates a temporary data set called ’pop’, just like 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 ‘missover’ 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.

‘Missover’ 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 ‘missover’ 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 'dlm' is used to indicate that the delimiter is an '!'. For tab delimiter data, use "dlm='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, especially 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 FY15 institutional TED SAS Dataset. When SAS executes this code, it reads in all 6 requested variables only when the 'where' statement is true.

```
libname intedi15 '/mdr/pub/tedi/fy15';

data tedi;
    set intedi15.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 large text files, use code such as this:

```
filename in pipe "uncompress –c /mdr/pub/deers/detail/vm6ben/fy10/fm12.txt.Z";

    data deers;
        infile in lrecl=8700 missover;
        input @307 primary $char1. @;          /*Need the "@" symbol when doing a
                                                 conditional read*/
        if (primary eq '1') then do;
            input @297 mhselig $char1.
                  @495 edipn $char10.;         /*Must have a second INPUT statement
                                                when performing a conditional read*/
            end;
            else delete;                       /* Only those conditions meeting the
                                                condition are kept*/
    run;
```

The only difference is the way that input statement is written. 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.

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 but this document will only touch on some of the more common 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.fy16(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.

Do End

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'.

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 */
```

You must have an '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 service not in ('X' 'Z');

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 Catchment Area Directory (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 00010001000100010404040404 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.fy16.txt";        /* name of format is $par16X. */
                                                  /* layout of format in MDR spec */
data temp;
        set in.fy16(keep=enrmtf);
        length mtfstring $50;
        mtfstring=put(enrmtf,$par16x.); /*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 enrtreg $2;
        enrsvc=substr(put(enrmtf,$par16X.),39,1); /* Branch of service */
        enrreg=substr(put(enrmtf,$par16X.),31,2); /* Region – 01, 02, ... */
        enrtreg=substr(put(enrmtf,$par16X.),40,1); /* HSSC region – N, S, W, O */
```

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

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,$par16X.),39,1);
mtfsvc=substr(put(mtf,$par16X.),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 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);	09 (character)
day=substr(caldate,7,2);	12 (character)
yr=substr(caldate,1,4);	1989 (character)
sasdate=mdy(mon,day,yr);	10847
date1=input(caldate,yymmdd8.);	10847
sasdate="12SEP1989"d;	10847
mon=month(sas date);	9 (number)

day=day(sas date);	12 (number)
yr=year(sas date);	1989 (number)

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

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

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 Risk Adjustment File for FY16:

```
options nocenter;
  libname riskfy16 '/mdr/pub/riskadjustment/fy16';
  PROC CONTENTS data=riskfy16.health_risk;
  run;
```

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

1. Open PICO at the command line once in the desired directory (type ‘pico’)
2. Type in program (above)
3. Hit [cntl X] to save program and name the program (Include .sas at the end of the name)
4. Run program using the SAS command (sas contents.sas)
5. Review the SAS Log (more contents.log)
6. Review the output from the procedure (more contents.lst)

The list file contains the output from ‘proc contents’.

At command prompt in Putty type: pico contents.lst to see the full output.

An example is:

```
The SAS System                                                          07:43 Wednesday,

The CONTENTS Procedure

Data Set Name        RISKFY16.HEALTH_RISK                    Observations          11050315
Member Type          DATA                                    Variables             138
Engine               V9                                      Indexes               0
Created              Tuesday, July 19, 2016 05:13:18 PM      Observation Length    1029
Last Modified        Tuesday, July 19, 2016 05:13:18 PM      Deleted Observations  0
Protection                                                   Compressed            BINARY
Data Set Type                                                Reuse Space           NO
Label                                                        Point to Observations YES
Data Representation  HP_UX_64, RS_6000_AIX_64, SOLARIS_64, HP_IA64  Sorted        YES
Encoding             latin1  Western (ISO)

                      Engine/Host Dependent Information

Data Set Page Size          32768
Number of Data Set Pages    34928
Number of Data Set Repairs  0
Filename                    /mdr/pub/riskadjustment/fy16/health_risk.sas7bdat
Release Created             9.0301M2
Host Created                AIX
Inode Number                327697
Access Permission           rw-rw----
Owner Name                  madm
File Size (bytes)           1144528896

                Alphabetic List of Variables and Attributes

  #    Variable      Type    Len    Label

134    ACV           Char      1    ALTERNATE CARE VALUE
126    AGEGRP        Char      1    AGE GROUP CODE
118    AGE_65        Char      1    65 AND OLDER FLAG
125    ASSIGN_UIC    Char      8    ASSIGNED UNIT
124    ATTACH_UIC    Char      8    ATTACHED UNIT
127    BENCAT        Char      3    BENEFICIARY CATEGORY
136    CATCH         Char      4    CATCHMENT AREA
```

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 riskfy16 '/mdr/pub/riskadjustment/fy16';
proc contents data=riskfy16._all_;
run;
```

This will yield the following:

```
The CONTENTS Procedure

                  Directory

Libref             RISKFY16
Engine             V9
Physical Name      /mdr/pub/riskadjustment/fy16
Filename           /mdr/pub/riskadjustment/fy16
Inode Number       327695
Access Permission  rwxr-xr-x
Owner Name         madm
File Size (bytes)  256


                   Member
#  Name            Type      File Size  Last Modified

1  HEALTH_RISK     DATA     1144528896  20Jul16:13:08:03
2  RISK_ADJUSTMENT DATA     2045779968  20Jul16:13:08:23
```

After the list of file size and last modification, a list of all variables in each file is listed.

'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 insidr15 '/mdr/pub/sidr/fy15/sidr.fy15';
data inpat;
        set insidr15.fy15(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 intedi "/mdr/pub/tedi/fy15";
data tedi;
        set intedi.header(keep = fy fm tedno bencat adm);
        where bencat = 'ACT';
run;
proc print data = tedi (obs = 10);
        var tedno;
        title "Sample of 10 records from TEDI";
run;
```

The SAS list file would contain 10 observations of the variable tedno from the SAS dataset 'tedi'. If no distinct variables were specified in the print statement, all 5 variables from the Keep statement in the Data step would be printed (fy fm tedno bencat adm)).

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 tu 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;

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

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

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;
```

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! '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>;

This example program tabulates population by gender.

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

'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.

'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.

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 dmissex pop
1 F 720
2 M 704

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 insidr15 '/mdr/pub/sidr/fy15/sidr.fy15';

data a;
        set insidr15.fy15; /* All FY15 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 ‘name’ which is then applied in the program to the variable ‘dmisid’.

```
options nocenter ps=32000 ls=132 obs=max;

libname in "/hpae2/kennell/veronika/wendy/jif";

proc format;
        value $name
          '0006'='JB Elmendorf/Richardson'
          '0014'='Travis AFB'
          '0024'='Camp Pendleton'
          '0048'='Fort Benning'
          '0049'='Ft Stewart'
          '0057'='Ft Riley'
          '0067'='WRNMMC'
          '0091'='Camp Lejeune'
          '0117'='Lackland AFB'
          '0437'='Schofield Barracks';
run;

data wt1a;
        set in.mtf_file (keep = fy fm dmisid……);
        length mtf_name $30.;
        mtf_name=put(dmisid,$name.);
run;
```

Format file are very useful when a programmer wants to ensure a common variable is covered in the same accepted way across multiple programs. This concept is especially useful when considering the DMISID format in the MDR 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, such as in the example above. 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:

```
options nocenter ls=132 ps=32000 obs=max;
libname out "/hpae2/kennell/tracy/Wendy";

data out.append;
   set out.urirxsumm3
       out.sinufinl;
run;
```

The members 'urirxsumm3' and 'sinufinl' from the 'out' SAS library are appended to one another in a dataset called 'append'. This procedure can be used on temporary datasets (outside of a standing library, such as "out"). The code would be structured in the same way as the code above, just without the "out" library name.

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

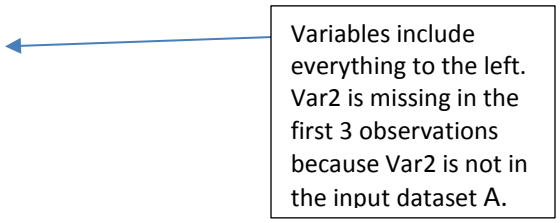
OBS	Type	Var1
1	X	A
2	Y	B
3	Z	C

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



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

PERS	DCCOST
1	3400
2	450
4	230

Purchased

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 (CAPER) 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. <u>You cannot mix character and numbers in an array.</u> 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 x_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: Flag PTSD CAPER encounters at DMIS ID 0052 in MEPRS code BFDR during FY15 FM12

```
options nocenter ls=132 ps=32000 obs=max;
libname out "/hpae2/kennell/tracy/Wendy";
libname incaper "/mdr/pub/caper/enhanced";

data caperset;
        set incaper.fy15(keep=fm comben dmisid apptidno meprscd dx1-dx3);
        where fm EQ '12' and dmisid EQ '0052' and comben EQ '4' and
        meprscd EQ 'BFDR';
run;

data out.array1;
        set caperset;
        length ptsd $1;
        array x_dx[3] $7. dx1-dx3;
        ptsd='N';
        do i=1 to 3;
```

```
            If substr(x_dx[i],1,5)='30981' then ptsd='Y';
        end;
        drop i;
run;

proc print data=out.array1 (obs=10);
        title "One Dimensional Array";
run;
```

Output:

ObsNum	DMIS ID	MEPRSCD	FM	COM BEN	APPTIDNO	DX1	DX2	DX3	PTSD
1	0052	BFDR	12	4	123	V6289	30981	33394	Y
2	0052	BFDR	12	4	456	V6229	30000		N
3	0052	BFDR	12	4	789	30000			N

Notional apptidno data

Only one variable was created in the above example but creation of more than one variable is also possible. In fact you can use an array to include additional variables. For instance, maybe we want to look at which diagnosis code contained the PTSD diagnosis.

```
data out.array2;
        set caperset;
        array x_dx[3] $7. dx1-dx3;
        array x_ptsd[3] $1. ptsd1-ptsd3;
        do i=1 to 3;
            If substr(x_dx[i],1,5)='30981' then x_ptsd[i]='Y';
            else x_ptsd[i]='N';
        end;
        drop i;
run;
```

Output:

ObsNum	DMIS ID	MEPRS CD	FM	COM BEN	APPTID NO	DX1	DX2	DX3	PTSD1	PTSD2	PTSD3
1	0052	BFDR	12	4	123	V6289	30981	33394	N	Y	N
2	0052	BFDR	12	4	456	V6229	30000		N	N	N
3	0052	BFDR	12	4	789	30000			N	N	N

In this case, the array x_ptsd[i] contains the variables ptsd1, ptsd2, and ptsd3.

Below are some reminders when using arrays:

- 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=‘15’;
%let dir=‘extract’;
libname in1 “/hpae2/kennell/laura/&dir./data”;  /* replace ‘&dir’ with ‘extract’
                                                    /hpae2/kennell/laura/extract/data*/
libname intedi “/mdr/pub/tedi/fy&fy2.”;         /* replace ‘&fy2’ with ‘15’ /mdr/pub/tedi/fy15*/

data temp;
        set in1.header;
        infile_fy=”FY&fy2.”; /*replace ‘&fy2 with 15  infile_fy=“FY15”*/
run;
```

‘%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/tracy/Wendy/uriapptlst2.txt';←used to reference a format file
```

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 common. 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

```
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;
run;

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

Use the %macro and %mend (next page) statements together!

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

%mend read_vm6;
%read_vm6(fy2=10,fm2=01)
%read_vm6(fy2=10,fm2=02)
%read_vm6(fy2=10,fm2=03) ←--when calling macros, do not have to include ';' at the end
%read_vm6(fy2=10,fm2=04)
%read_vm6(fy2=10,fm2=05)
%read_vm6(fy2=10,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=10,fm2=01) , whenever the line '&fy2' is seen, it will be replaced by the parameter in the call statement (in this case '10'). Whenever the line '&fm2' is seen, it will be replaced by the second parameter in the call statement (in this case '01').

```
filename invm6 pipe "uncompress -c /mdr/pub/deers/detail/vm6ben/fy10/fm01.txt";

data deers1001;
        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;
run;

proc append data= deers1001 base=all_deers;
run;


proc datasets nolist;
        delete deers1001;
run;
```

3. In the 2nd macro call %read_vm6(fy2=10,fm2=02) , whenever the line '&fy2' is seen, it will be replaced by the parameter in the call statement (in this case '10'). 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/fy10/fm02.txt";

data deers1002;
        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;

run;

proc append data= deers1002 base=all_deers;
run;

proc datasets nolist;
        delete deers1002;
 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 ‘deers1001’ dataset (containing fy08, fm01 data). After the second iteration, SAS would append the dataset ‘deers1002’ 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 ‘deers1002’ 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 "/hpae2/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.

Example:

```
filename out1 "/hpae2/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 /hpae2/kennell/
At the command prompt type: more enrsum.txt

File enrsum.txt will look like this:
OCT150110023
OCT1502957
OCT15033344
OCT150487

Delimited Files: Writing delimited files is 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 "/hpae2/kennell/enrsum.txt";

data _null_;
        set enroll;
        file out1 lrecl=30 dlm= '!';
        put @1 mon_yr region enr_ct;
run;
```

At the command prompt type: cd /hpae2/kennell/
At the command prompt type: more enrsum.txt

File enrsum.txt will look like this:

OCT15!01! 10023
OCT15!02! 957
OCT15!03! 3344
OCT15!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 (e.g. 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 except (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

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

There are five types of errors in SAS code:

- Syntax errors: when statements do not conform to the rules of the SAS language
- Semantic errors: when the structure of statement is incorrect, or an element is incorrectly specified
- Execution-time error: when the code is compiled correctly, but the data doesn't work with that code (e.g. division by zero)
- Data error: when data values do not match INPUT statement specification
- Logic error: when the code runs correctly, but does not give the expected results

Most of these errors can be resolved using a program's log. The log contains warnings and other important information. Logs should ALWAYS be read through! Do searches (ctrl w) on the words 'error',

'warn', 'uninitialized', and/or 'invalid' as they will be in the logs if the programs had errors. Logs can also be used to keep careful track of data as it flows throughout a program. Note record counts in each data step. Review the numbers and be sure they make sense.

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.

Good luck using the MDR!

Appendix A: Sample MDR User Profile Set-Up

```
#PATH=/usr/bin:/etc:/usr/sbin:/usr/ucb:$HOME/bin:/usr/bin/X11:/sbin
PATH=$PATH:/usr/bin:/etc:/usr/sbin:/usr/ucb:$HOME/bin:/usr/bin/X11:/sbin:/usr/local/bin

PS1='[$PWD]> '

export PATH PS1 TMOUT

if [ -s "$MAIL" ]           # This is at Shell startup.  In normal
then echo "$MAILMSG"        # operation, the Shell checks
fi                          # periodically.

umask 077

CHARMHOMEDIR=/hpae2/kennell/util
export CHARMHOMEDIR

alias ll='ls -lrt'
alias jobs='ps -ef | grep -i sasexe'
alias keith='cd /hpae2/kennell/keith'
alias pub='cd /mdr/pub'

export MCRBEGFY4=2001
export MCRBEGFY2=01
export MCRFY2=1
export MCRFY4=2014
export MCRLVMFILE2014=/mdr/pub/deers/lvm/lvm6/fy14.txt.Z
export MCRLVMFILE2013=/mdr/pub/deers/lvm/lvm6/fy13.txt.Z
export MCRLVMFILE2012=/mdr/pub/deers/lvm/lvm6/fy12.txt.Z
export MCRLVMFILE2011=/mdr/pub/deers/lvm/lvm6/fy11.txt.Z
export MCRLVMFILE2010=/mdr/pub/deers/lvm/lvm6/fy10.txt.Z
export MCRLVMFILE2009=/mdr/pub/deers/lvm/lvm6/fy09.txt.Z
export MCRLVMFILE2008=/mdr/pub/deers/lvm/lvm6/fy08.txt.Z
export MCRLVMFILE2007=/mdr/pub/deers/lvm/lvm6/fy07.txt.Z
export MCRLVMFILE2006=/mdr/pub/deers/lvm/lvm6/fy06.txt.Z
export MCRLVMFILE2005=/mdr/pub/deers/lvm/lvm4/fy05.txt.Z
export MCRLVMFILE2004=/mdr/pub/deers/lvm/lvm6/fy04.txt.Z
export TODAY=date
```

Names assigned to commands and directories most often used

Appendix B: Access Permissions

1. Permissions

a. There are default file permissions set in UNIX, but these can be modified.

i. Access permissions are assigned to three different entities:

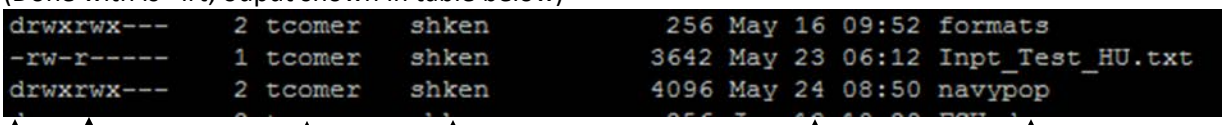
a) Owner: the one who creates the file (has full permission as default)
b) Group: the folks who are in your group (e.g. company)
c) Other: others who have access to the root files in a directory (those outside group).

ii. Types of Access:

a) Read
b) Write
c) Execute (means you can "run" a program)

Example of output with read/write permissions:

(Done with ls –lrt; ouput shown in table below)



Type Permissions owner group file size in bytes date last updated dir or file name

iii. The access permissions can be described by separating into components:

a) Type (e.g. d or -)
b) Owner- 1st 3 digits of permissions (e.g. rwx= read, write, execute)
c) Group- 2nd group of 3 digits of permissions (e.g. rwx= read, write, execute)
d) Other- 3rd group of 3 digits of permissions (e.g. ---= no priveleges)

2. Setting and Changing Permissions

a. Use the UNIX command CHMOD to change access permissions by summarizing the numerical values for Owner, Group and Other.

Digit	Permissions
0	None
1	Execute only
2	Write only
3	Write and execute

4	Read only
5	Read and execute
6	Read and write
7	Read, write, and execute

b. chmod in Action

i) Example 1: chmod 770 tedn.sas - allows owner and group to have read/write/execute privileges, other no privileges

ii) Example 2: chmod 764 tedn.sas - allows owner read/write/execute, allows group read/write only, allows other read only

Appendix C: Additional UNIX Commands

Functional Category	Command	Use	Example	Example Explanation
Modifying directories and files	rm	Delete files	rm data/*	Removes all files in the directory called data
	rmdir	Delete directories (directory must be empty)	rmdir data	Deletes the directory called data
Space usage & Reading, editing, and getting info about text files	du	Shows disk usage of a directory or file	du -g .	Shows the data usage (in GB) of the files and directories in the current directory
	more	Open one page of a file at a time. Use Enter to navigate forward one line at a time and Space to navigate forward one page at a time. Useful for examining large files.	more sample.txt	Opens up the first page of sample.txt for viewing
	head	View first 10 lines of a file. Useful for examining large files.	head sample.txt	Prints first 10 lines of sample.txt
	tail	View last 10 lines of a file. Useful for examining large files. Can be used in tandem with head to look at sections of a file besides the beginning and end.	head -6 sample.txt | tail -3	Prints lines 4-6 of sample.txt (the number after head gives what line to stop at, the number after tail gives how many lines to read)
	diff	Compares files, and shows where they differ	diff sample.txt sample2.txt	Shows where these two files differ
	wc	Tells you how many lines, words, and characters there are in a file	wc sample.txt	Returns the line, word, and character counts in sample.txt
	mdata	Get metadata for a file (typically used for files in /mdr/pub). See Section A-9 in the MDR Users Guide.	mdata champus.sas7bdat	Returns the metadata of the TEDN CHAMPUS file

Appendix D: Comparison of MDR 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>.sas7bdat	Limit to records where Primary Record Flag=1 and MHS Eligibility Flag=1. Monthly text files available for data prior to FY11. All months but the most recent are compressed. Need to use special UNIX commands when reading in compressed text 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/Dental/Direct Care Dental	/mdr/pub/dental/ded/fy<fy>.sas7bdat	
Health Care Services/Dental/Purchased Care Dental	/mdr/pub/dental/addp/claims/fy<fy>/fy<fy>.sas7bdat /mdr/pub/dental/tdp2/claims/fy<fy>.sas7bdat	
Health Care Services/Direct Care/Ancillary Services/Radiology Detail	/mdr/pub/ancillary/fy<fy>/ancillary.fy<fy>/fy<fy>.sas7bdat	Use Record Type to separately identify Radiology records. Rectype='R'
Health Care Services/Direct Care/Ancillary Services/<FY> Lab Detail	/mdr/pub/ancillary/fy<fy>/ancillary.fy<fy>/fy<fy>.sas7bdat	Use Record Type to separately identify lab records. Rectype='L'
Health Care Services/Direct Care/Appointments	/mdr/pub/appt/detail/fy<fy>/fy<fy>.sas7bdat /mdr/pub/appt/detail/fy<fy>/fy<fy>_cancel.sas7bdat	
Health Care Services/ Direct Care / CDR Vitals	/mdr/pub/cdr/vitals/fy<fy>.sas7bdat	
Health Care Services/Direct Care/Inpatient Admissions	/mdr/pub/sidr/fy<fy>/sidr.fy<fy>/fy<fy>.sas7bdat	
Health Care	/mdr/pub/caper/enhanced/fy<fy>.sas7bdat	

Services/Direct Care/<FY> Professional Encounters		
Health Care Services/Purchased Care/Institutional	/mdr/pub/tedi/fy<fy>/header.sas7bdat	
Health Care Services/Purchased Care/NonInstitutional/ 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/NonInstitutional/ 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	
Health Care Services/Pharmacy/PDTS <FY>	/mdr/pub/pdts/detail/fy<fy>/fy<fy>.sas7bdat	
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	Available through FY13.
Reference Tables/APC Codes	/mdr/ref/caper.apc.cy<cy>.cq<cq>.txt	
Reference Tables/CPT/HCPCS Codes	/mdr/ref/rvu.cy<cy>/rvumast.sas7bdat	
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/MS-DRG Codes	/mdr/ref/msdrgref.fy<fy>.txt	
Reference Tables/ICD-9 and 10 CM Diagnosis Codes	/mdr/ref/icd9dxref.fy<fy>.txt /mdr/ref/icd10dxref.fy<fy>.txt	
Reference Tables/ICD-9 and 10 CM Procedure Codes	/mdr/ref/icd9procref.fy<fy>.txt /mdr/ref/icd10procref.fy<fy>.txt	
Reference	/mdr/ref/cad.omni.fy<fy>.sas7bdat	

Tables/Market Area Table		
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/Skill Type Codes	/mdr/ref/eas4.skill.fy<fy>/fy<fy>.sas7bdat	
Reference Tables/Purchased Care Providers	/mdr/pub/tedpr/tedpr.sas7bdat	
Reference Tables / NPI Directory	/mdr/pub/nppes/nppes.sas7bdat	

Appendix E: Equivalent Variable Names Across Data Sets

Variable	Values	DEERS LELG	DEERS VM6BEN text	DEERS VM6BEN SAS data set	DEERS VM6ENR	TEDI	TEDN
Record Key		patuniq	N/A*	N/A*	patuniq	tedno	tedno || linum
FY	e.g. 2016	N/A	N/A	N/A	fy	fy	fy
FM	e.g. 01	N/A	N/A	N/A	fm	fm	fm
Begin Date		N/A	N/A	N/A	N/A	begdate	begdate
End Date		N/A	N/A	N/A	N/A	enddate	enddate
EDI_PN		patuniq	@495-504	DOD_EDI_PN_ID	patuniq	edi_pn	edi_pn
Sponsor SSN		sponssn	@9-17	SPN_PN_ID	sponssn	sponssn	sponssn
Patient SSN		N/A	@24-32	PN_ID	benssn	patssn	patssn
Age		N/A	@287-289	D_AGE_QY	age	patage	patage
Age Group	A-H, X	N/A	@286	D_AGE_GROUP_CD	dagegrp	agegrp	agegrp
Gender	F, M, Z	sex	@43	PN_SEX_CD	gender	patsex	patsex
Sponsor Service	A, C, F, H, M, etc.	N/A	@103	SVC_CD	N/A	N/A	N/A
Sponsor Service, Aggregate	A, C, F, M, N, V, etc.	svc	@306	D_SPON_BR_SVC_CD	service	dsponsvc	dsponsvc
Bencat	ACT, DA, etc.	ben01-ben12	@290-292	R_BEN_CAT_CD	dbencat	bencat	bencat
Comben	1, 2, 3, 4	comben	@309	D_COM_BEN_CAT_CD	comben	comben	comben
Medicare Eligibility	A, B, C, N	N/A	@310	D_MDC_ELIG_CD	dmedcare	tflflag*	tflflag*
ACV	A, B, E, F, H, etc.	acv01-acv12	@539	MDR_ACV	acv	acv	acv
ACV Group	Prime, Plus, etc.	N/A	N/A	N/A	N/A	acvgroup	acvgroup
Enrollment Site	DMIS ID	enr01-enr12	@447-450	D_MI_PCM_EDVSN_DMIS_ID	dmisid	denrsite	denrsite
Enrollment Service	A, F, M, N, etc.	N/A	N/A	N/A	N/A	enrsvc	enrsvc
Enrollment HSSC Region	N, S, W, A, O	dhsscreg	@562	D_HSSC_ENR_RGN_CD	dhsscenr	enrhssc	enrhssc
Treatment DMIS ID	DMIS ID	N/A	N/A	N/A	N/A	N/A	N/A
Treatment Service	A, B, F, G, N, R, etc.	N/A	N/A	N/A	N/A	N/A	N/A
Treatment HSSC Region	N, S, W, A, O	N/A	N/A	N/A	N/A	N/A	N/A
Patient ZIP Code		dzipcd	@301-305	D_ZIP_CD	dzipcd	deerszip	deerszip
Catchment Area ID	DMIS ID	dcatch	@279-282	D_CATCH_AREA_CD	dcatch	catch	catch
PRISM ID	DMIS ID	dprism	@293-296	D_PRISM_CD	dprism	prism	prism
Residence T3 Region*	N, S, W, A, O	N/A	@766**	D_T3_RES_REGION_CD**	N/A	resreg	resreg
Residence HSSC Region	N, S, W, A, O	N/A	@561	D_HSSC_RES_RGN_CD	N/A	restnex	restnex
Diagnoses		N/A	N/A	N/A	N/A	dx1-dx25	dx1-d25
Procedure Codes		N/A	N/A	N/A	N/A	proc1-proc25	cpt
MS-DRG		N/A	N/A	N/A	N/A	msdrg	N/A
MDC		N/A	N/A	N/A	N/A	mdc	mdc
Cost fields		N/A	N/A	N/A	N/A	paid, allow, ohi	paid, allow, ohi
Workload fields		N/A	N/A	N/A	N/A	adm, days, msrwp	svcs, enc, totrvu
Other useful variables		num_mons_elg, num_mons_enr	MHS Eligibility (@297), Primary Record (@307), Risk Score (@730-737)	D_MHS_ELIG_INDIC, D_PRIMARY_RECORD_FLAG, D_RISK_TRUNC_NO	risk_trunc_no	acute, insttype, admsrc, dispstat, admtedno	admdate, provspec, place, typsvc1, typsvc2, admtedno
*In all data sets besides VM6BEN, FY12+ = T3 Region. Before that, 01-16, AK.			*EDI_PN after taking primary records **FY11 forward	*EDI_PN after taking primary records **FY11 forward		*A, N, U, T	*A, N, U, T

Variable	Values	SIDR	CAPER	PDTS SAS data set	Ancillary	Vitals	MEPRS
Record Key		mtf || prn	hostdmis || apptidno	authnum	chcsdmis || accessno || cpt || cptmod	cdr_vitals_id	fy || fm || repdmis || mepr4
FY	e.g. 2016	fydisp*	fy	fy****	N/A	fy	fy
FM	e.g. 01	fmdisp*	fm	fm****	N/A	fm	fm
Begin Date		admdate	encdate**	datedisp	servdate	apptdt	N/A
End Date		dispdate	N/A	N/A	N/A	N/A	N/A
EDI_PN		patuniq	edipn	upid	edipn	edi_pn	N/A
Sponsor SSN		sponssn	sponssn	sponssn	sponssn	sponssn	N/A
Patient SSN		patssn	patssn	N/A	N/A	patssn	N/A
Age		recage	patage	age	patage	patage	N/A
Age Group	A-H, X	dmisage	agegrp	agegrp**	N/A	agegrp	N/A
Gender	F, M, Z	dmissex	patsex	gender	sex	gender	N/A
Sponsor Service	A, C, F, H, M, etc.	ssvclvm4	ssvclvm	svc	svc	dsponsvc	N/A
Sponsor Service, Aggregate	A, C, F, M, N, V, etc.	sagglvm4	sagglvm	d_spnsvc	svcagg	dsvcagg	N/A
Bencat	ACT, DA, etc.	bencatx	bencatx	r_bencat	bencat	bencat	N/A
Comben	1, 2, 3, 4	combenf	comben	comben	comben	comben	N/A
Medicare Eligibility	A, B, C, N	medelig2	medelig2	medflag	medelg	N/A	N/A
ACV	A, B, E, F, H, etc.	acv	acv	acv	acv	acv	N/A
ACV Group	Prime, Plus, etc.	N/A	acvgroup	acvgrp	acvgroup	N/A	N/A
Enrollment Site	DMIS ID	deersenr	enrdmis	enrdmis	enrdmis	denrsite	N/A
Enrollment Service	A, F, M, N, etc.	N/A	N/A	N/A	N/A	N/A	N/A
Enrollment HSSC Region	N, S, W, A, O	N/A	enrreg*	enrreg*	enrreg	N/A	N/A
Treatment DMIS ID	DMIS ID	mtf	dmisid	txdmis	tmtdmis	dmisid	repdmis
Treatment Service	A, B, F, G, N, R, etc.	mtfsvc	txsvc	N/A	tmtsvc	N/A	N/A
Treatment HSSC Region	N, S, W, A, O	mtfhssc	txhssc	rx_reg***	tmtreg	N/A	N/A
Patient ZIP Code		patzip	patzip***	d_zip	patzip	deerszip	N/A
Catchment Area ID	DMIS ID	catch	catch	d_catch	catch	N/A	N/A
PRISM ID	DMIS ID	prism	prism	d_prism	prism	N/A	N/A
Residence T3 Region*	N, S, W, A, O	patregn	patregn	d_region	patregn	N/A	N/A
Residence HSSC Region	N, S, W, A, O	pathssc	pathssc	hsscreg	rsreg	N/A	N/A
Diagnoses		dx1-dx20	dx1-dx10	N/A	N/A	N/A	N/A
Procedure Codes		proc1-proc20	cpt_1-cpt_13	N/A	cpt, cptmod	N/A	N/A
MS-DRG		msdrg	N/A	N/A	N/A	N/A	N/A
MDC		msmdc	mdc	N/A	N/A	N/A	N/A
Cost fields		fullcost	fcost, fcrx	ingrcost, netamtpd	N/A	N/A	mtfexp
Workload fields		dmisdays, msdrgrwp	rvu_et, trvuagg, apcaggwt	daysuply, decqty	drvcount, rvu	N/A	assigfte, availfte
Other useful variables		infflag, admsrc, disptype, icudays	apptinfr, meprscd, dispcode, mtf_prn	fillloc, ndc, prodname, gcn, therclss, orddmis, apptien	rectype, orddmis, apptno, prn	hostdmis, apptidno	mepr4
*In all data sets besides VM6BEN, FY12+ = T3 Region. Before that, 01-16, AK.		*Stored as numeric	*00-15, AK **Stored as character ***9-digit ZIP	*00-15, AK **A-L ***T,R,V: blank FY12+: C+D: 00-15, AK; M: N, S, W, A, O. before: C,D,M: 01-16, AK ***FY10 forward			

Appendix F: Sample Programs

Example 1- Assigning Logs and Lists to a Separate Folder

```
*Add this code at the beginning of your program to write out the logs and lsts to a separate folder;
*This code will add a date and time stamp, so that you'll have a unique record of each time the program
was run;

%let dir1=%str(/hpae2/kennell/training/examples/log/);
%let pgmname=%str(lahproc_contents.sas);
%let date=&sysdate;
%let time=&systime;
%let ext1=%str(log);
%let break=%str(-);
%let final1=&dir1&pgmname&date&break&time..&ext1;
%put &final1;
proc printto log="&final1";
run; quit;
```

Example 2 – Proc Contents

```
*
*      PROGRAM NAME  : proc_contents.sas
*      DESCRIPTION   : Proc Contents Example
*      PROGRAMMER    : Kennell
*;

options nocenter ls=132 ps=32000 obs=max;

libname sidr '/mdr/pub/sidr/fy13/sidr.fy13/';

proc contents data=sidr.fy13;
run;
```

Example 3 – Reading in Data Using Conditional Statements

```
*
*          PROGRAM NAME  : tedi.sas
*          DESCRIPTION   : TED- I Example
*          PROGRAMMER    : Kennell
*;

options nocenter ls=132 ps=32000 obs=max;
```

```
libname in "/mdr/pub/tedi/fy13";

*Read in TEDI file;
data tedi;
	set in.header(keep = denrsite mdc acute comben insttype dispstat msrwp acv msdrg paid);

*Use the where statement to conditionally read in a data set;
*This is the most efficient way to narrow down data to only what you are interested in;
*Remove Interim Claims;
*Interim Claims have dispstat = '30', which means that the patient is still in the hospital (i.e. have not
been discharged);
	where dispstat ne '30';

*Derive Disposition counter;

*In the TEDI, every record (after removing interim claims) represents a disposition;
	disposition = 1;

*Derive MDC Group;
	length mdcgroup $3.;
	if mdc in ("14" "15") then mdcgroup = "OB";
	else if mdc in ("19" "20") then mdcgroup = "MH";
	else mdcgroup = "OTH";

*Derive Enrollment Status;
*This uses nested if/then/else/do statements;
*To be Civ Prime, the record must meet the condition "if acv in ('A' 'E' 'H' 'J')" and
*   the condition "if substr(denrsite,1,2) in ('69' '79')";
	length enrstat $9.;
	if acv in ('A' 'E' 'H' 'J') then do;
		if substr(denrsite,1,2) in ('69' '79') then enrstat = 'Civ Prime';
		else enrstat = 'MTF Prime';
	end;
	else enrstat = 'Other';
run;

proc freq data = tedi;
	tables acute*insttype /missing list;
	title "Acute and Non-Acute Care by Institution Type";
run;

proc freq data = tedi;
	tables enrstat /missing list;
	title "Dispositions by Enrollment Status";
run;

*Find Case Mix for each MDC Group;
proc summary data = tedi nway;
```

```
        class mdcgroup /missing;
        var msrwp disposition;
        output out = tedisum1 sum=;
run;

data tedisum1;
        set tedisum1;

*Derive Case Mix;
        case_mix = msrwp/disposition;
run;

proc print data = tedisum1;
        title "Case Mix by MDC Group";
run;

proc datasets;
        delete tedisum1;
run;

*Find top 5 MS-DRGs by Amount Paid;
proc summary data = tedi nway;
        class msdrg /missing;
        var paid;
        output out = tedisum2 sum=;
run;

proc sort data = tedisum2;
        by descending paid;
run;

proc print data = tedisum2 (obs = 5);
        title "Top 5 MS-DRGs by Amount Paid";
run;

proc datasets;
        delete tedi tedisum2;
run;
```

Example 4- Using a Proc Format and Reading in 2 Data Sets

```
*
*       PROGRAM NAME  : sidr.sas
*       DESCRIPTION   : SIDR Example
*       PROGRAMMER    : Kennell
*;
```

```
options nocenter ls=132 ps=32000 obs=max;

libname in13 "/mdr/pub/sidr/fy13/sidr.fy13/";
libname in12 "/mdr/pub/sidr/fy12/sidr.fy12/";

PROC FORMAT;
   VALUE $mdc
'19','20'="MH"
'14','15'="OB"
other="OTH";
run;

/* Read in all years of SIDR and do a frequency by FY */
        data sidrall;
         set in12.fy12(keep=fydisp fmdisp) in13.fy13(keep=fydisp fmdisp);

         proc freq data=sidrall; tables fydisp/list missing;
         title 'Frequency of SIDR records by FY';

/* Read in FY13 SIDR and demonstrate various programming techniques */

        data sidr;
                set in13.fy13(keep = fydisp fmdisp calcdays msdrg mtfsvc acv msmdc mtf deersenr

                catch bencatx combenf dmisage fullcost dispdate msdrgrwp infflag);

                        disposition = 1;

                /* Create new variable called MDC Group, which will group records into 3 categories:
OB, MH, and Other */
                /* This method uses the if/then/else concept */
                /* It is a good idea to set the length of the variable prior to creating it otherwise the
length will */
                /* automatically be set to the first value, in this case 2 */

                        length mdcgroup $3.;
                        if msmdc in ("14" "15") then mdcgroup = "OB";
                                else if msmdc in ("19" "20") then mdcgroup = "MH";
                                else mdcgroup = "OTH";

                /* An alternate way to create a new variable is to use proc format*/
                /* This method searches for a value of a certain variable and then replaces it with what's
in the format */
                /* Similar concept to VLOOKUP in Excel */

                        mdcgroup2=put(msmdc,$mdc.);
```

```
/* Run Proc Freq to generate statistics on various variables */
/* Note that you can do more than one frequency at a time */

        proc freq data=sidr; tables mdcgroup*mdcgroup2 mdcgroup*msmdc/list
missing;
        title 'Cross tab of 2 methods of grouping variables';


/* Generate cross tab of MDC Group by Beneficiary Category */

        proc freq data=sidr; tables mdcgroup*combenf/list missing;
        title 'Frequency of MDC Group by Common Beneficiary Category ';

/* Note that inferred SIDRs will have data like Beneficiary Category, but not MDC */

        proc freq data = sidr; tables infflag*combenf infflag*msmdc /list missing;
        title 'Bencat and MDC by Inferred SIDR Flag';

/* Determine top 10 MSDRGs based on Full Cost*/
/* First do a summary of full cost by MSDRG */
/* Then, sort data by fullcost, greatest to smallest */
/* Then, print 10 records */

        proc summary data = sidr nway; class msdrg/missing;
        var fullcost msdrgrwp disposition;
        output out=sidrsum sum=;

        proc sort data = sidrsum;  by descending fullcost;
        run;

        proc print data = sidrsum (obs=10); var msdrg fullcost;
        title "Top 10 MSDRGs by fullcost";
        run;

/* Calculate Case Mix */
        data sidrsum;
                set sidrsum;
                case_mix = msdrgrwp/disposition;

        run;

        proc print data = sidrsum(obs=10);
        title 'Print of Case Mix';

        run;
```

Example 5- Applying the DEERS format, Using Substring and Put Functions

```
 *
*          PROGRAM NAME  : caper.sas
*          DESCRIPTION   : CAPER Example
*          PROGRAMMER    : Kennell
*;

options nocenter ls=132 ps=32000 obs=max;

libname in "/mdr/pub/caper/enhanced";

*Include the DMIS ID Index Table Format File;
%include '/mdr/ref/dmisid.index.fy13.txt';

*Read in CAPER file;
data caper;
          set in.fy13(keep = meprscd comben acv mdc dx1 cpt_4 apptinfr fcost fcrx enrdmis);

*To get non-pharmacy costs, simply subtract fcrx from fcost;
          nonrxcost = fcost - fcrx;

*Derive Prime Enrolled flag using ACV;
*This method uses the if/then/else concept;
*It is a good idea to set the length of the new variable (prime);
          length prime $1.;
          if acv in ('A' 'E' 'H' 'J') then prime = 'Y';
          else prime = 'N';

*Derive MEPRS 1 Code using Substring command;
*The Substring command has three parameters: substr(par1,par2,par3);
*   par1 is the variable that you want to substring from (in this case meprscd);
*   par2 is the position you want to start reading from (in this case 1);
*   par3 is the length that you want to read (in this case 1);
*So in this case, we are taking the variable meprscd, starting at the 1st position and reading for 1
character;
*If the MEPRS Code was BAAA, this would return B;
          length meprs1 $1.;
          meprs1 = substr(meprscd,1,1);

*Apply the DMIS ID Format file using "put" and "substr" combined;
*Always define the length of the variables when doing this;
*If you do not, the default length will be the length of the format file, which is 246 bytes;
*The name of the format is $parYYx. where YY is the two digit Fiscal Year;
*We are going to feed the format the Enrollment Site, and return the Enrollment HSSC Region
(N,S,W,A,O);
          length enrhssc $1.;
          enrhssc = substr(put(enrdmis,$par13x.),40,1);
```

```
*CAPER is one of the few files that stores dates as character (rather than SAS dates);
*Input will always return a number, in this case a SAS date;
*The informat "yymmdd8." will read in any 8-digit character date that has year, then month, then date;

*      regardless of what separators might be used (dashes, slashes, or, in this case, no separator);


          sas_encdate = input(encdate,yymmdd8.);
run;


*Do frequencies on derived variables;
*Proc Freq will count the number of records in the dataset that have each value for the variables listed
after tables;
*You can do two dimensional (or more) frequencies by including a * between the variables of interest;

proc freq data = caper;
          tables prime*comben /missing list;
          title "Prime Enrollees by Comben";
run;

proc freq data = caper;
          tables meprs1 /missing list;
          title "MEPRS 1 Code";
run;

proc freq data = caper;
          tables enrhssc /missing list;
          title "Enrollment HSSC Region";
run;


*Get Total Costs for Inferred and Non-Inferred CAPERs, with and without Pharmacy Costs;
*Proc Summary is a good way to aggregate data to a higher level;
*In this case we aggregating from a record level file to a file with rows for each value of apptinfr (Y or N);

proc summary data = caper nway;
          class apptinfr /missing;
          var fcost nonrxcost;
          output out = capersum1 sum=;
run;

*Proc Print will display the data set in the .lst file;
proc print data = capersum1;
          title "Costs for Inferred and Non-Inferred CAPERs, with and without Pharmacy Costs";
run;

*Proc Datasets can be used to delete data sets that are no longer necessary;
```

```
*This frees up workspace so that your program will run faster;
proc datasets;
	delete capersum1;
run;


*Find top MDCs by Full Cost;
proc summary data = caper nway;
	class mdc /missing;
	var fcost;
	output out = capersum2 sum=;
run;

*Proc Sort puts the data set in order according to the variables listed in the by statement;
*You can sort from highest to lowest by using descending BEFORE the variable you want to go highest to
lowest;
proc sort data = capersum2;
	by descending fcost;
run;

proc print data = capersum2;
	title "Top MDCs by Full Cost";
run;

proc datasets;
	delete caper capersum2;
run;

*Analayze CAPERs with MDC = 23 (Factors Influencing Health Status and Other Contacts with Health
Services),
*PDHAs, and Case Management CAPERs;
*We will read in one data set (the CAPER) and output to three data sets (mdc23, pdha, and cm);
data mdc23 pdha cm;
	set in.fy13(keep = fy fm dx1 meprscd cpt_4 comben mdc);

*Derive Encounter counter;
*Each record in CAPER represents an encounter;
	enc = 1;

*Output CAPERs with MDC = 23 to the mdc23 data set;
	if mdc = '23' then output mdc23;

*Output CAPERs with primary diagnosis indicating PDHA to the pdha data set;
	if dx1 = 'V705 E' then output pdha;

*Output CAPERs that meet the criteria for Case Management to the cm data set;
	if substr(dx1,1,5) = 'V4989' and substr(cpt_4,1,1) = 'G' and
		meprscd in ('FAZ2' 'ELAN' 'ELA2') then output cm;
```

```
run;

*Find top 20 Primary Diagnoses in MDC23 by Encounters;
proc summary data = mdc23 nway;
        class dx1 /missing;
        var enc;
        output out = summdc23 sum=;
run;

proc sort data =summdc23;
        by descending enc;
run;

proc print data = summdc23 (obs=20);
        title "Top 20 Diagnoses in MDC 23 by Encounter";
run;

proc datasets;
        delete mdc23 summdc23;
run;

*Show trend in PDHAs, and which Beneficiary Categories are filling out PDHAs;
proc freq data = pdha;
        tables fy*fm comben / missing list;
        title "Trend in PDHAs over time, and by Comben";
run;

proc datasets;
        delete pdha;
run;

*QC criteria for Case Management CAPERs;
proc freq data = cm;
        tables fy*fm meprscd dx1 cpt_4 /missing list;
        title "Case Management CAPERs";
run;

proc datasets;
        delete cm;
run;
```

Example 6- Getting a Count of Unique Users

```
*       PROGRAM NAME  : tedn2.sas
*       DESCRIPTION   : Number of Unique MH Users
*       PROGRAMMER    : Kennell
*;
```

```
options nocenter ls=132 ps=32000 obs=max;

libname in "/mdr/pub/tedni/fy13";

*Read in TEDNI file;
data tedni;
        set in.champus(keep = edi_pn mdc denrsite tflflag in = a)
                in.tdefic(keep = edi_pn mdc denrsite tflflag in = b);
        if mdc in ('19' '20');

        if b and tflflag = 'U' then delete;
run;

proc sort data = tedni;
        by edi_pn;
run;

data tedni;
        set tedni;
        by edi_pn;

        if first.edi_pn then person = 1;
        else person = 0;
run;

proc summary data = tedni nway;
        class denrsite /missing;
        var person;
        output out = tedsum sum=;
run;

proc datasets;
        delete tedni;
run;

proc print data = tedsum;
run;
```

Example 7- Creating a Format File for Future Use in a Program

```
*       PROGRAM NAME  : continuous.sas
*       DESCRIPTION   : Find how many continuously enrolled/eligible and new enrollees etc.
*       PROGRAMMER    : Kennell
*;

options nocenter ls=132 ps=32000 obs=max;
```

```
libname inlelg "/mdr/pub/deers/lelg/vm6lelg";
filename out "/hpae2/kennell/training/examples/prime.fmt";

*Read in Longitudinal DEERS;

data lelg13 prime (keep=patuniq prime);
        set inlelg.fy13(keep=patuniq acv01-acv12 num_mons_elg num_mons_enr);


*Eligible in FY 13;
        if num_mons_elg > 0 then elg = 1;
        else elg = 0;

*Continuously eligible in FY 13;
        if num_mons_elg = 12 then contelg = 1;
        else contelg = 0;

*Eligible for 11 out of 12 months;
        if num_mons_elg = 11 then elg11 = 1;
        else elg11 = 0;

*Enrolled in FY 13;
        if num_mons_enr > 0 then enr = 1;
        else enr = 0;

*Continuously enrolled in FY 13;
        if num_mons_enr = 12 then contenr = 1;
        else contenr = 0;

*Enrolled for 11 out of 12 months;
        if num_mons_enr = 11 then enr11 = 1;
        else enr11 = 0;

*Check if prime enrolled for all of FY13;
        if acv01 in ('A' 'E' 'H' 'J') and
                acv02 in ('A' 'E' 'H' 'J') and
                acv03 in ('A' 'E' 'H' 'J') and
                acv04 in ('A' 'E' 'H' 'J') and
                acv05 in ('A' 'E' 'H' 'J') and
                acv06 in ('A' 'E' 'H' 'J') and
                acv07 in ('A' 'E' 'H' 'J') and
                acv08 in ('A' 'E' 'H' 'J') and
                acv09 in ('A' 'E' 'H' 'J') and
                acv10 in ('A' 'E' 'H' 'J') and
                acv11 in ('A' 'E' 'H' 'J') and
                acv12 in ('A' 'E' 'H' 'J') then prime = 'Y';
                else prime = 'N';
```

```
        if prime = 'Y' then output prime;

        drop acv01-acv12 num_mons_elg num_mons_enr;

        output lelg13;
run;

*There should not be any duplicates in the LELG data, but dedup just to be sure. The format file will not
work with duplicates;
proc sort data = prime nodupkey;
        by patuniq;
run;

*Output format file for those continuously prime enrolled in FY13;
data _null_;
        set prime end = last;

        file out;

        if _n_ = 1 then put         'proc format; ' /
                                    '           value $prime';

        if last then put @ 10       ' patuniq $10. "'=" "'Y'" /
                         @10        'other = "N";' /
                         @1 'run;';

        else put         @10        ' patuniq $10. "'=" "'Y'";
run;

*The previous format file cannot be used in this same program, but a permanent version of it is saved to
the filename detailed above;
*If you wanted to make the same format file, and use it in the same program, follow the steps below;

*Note, however, that the method below will not create a permanent version. The format can only be
used in this program;

data prime;
*Whatever you want on the left side of the format, rename as start;
        set prime(keep = patuniq rename = patuniq=start) end=last;

        length label hlo type $1. fmtname $6.;

*The "fmtname" is how you will call the format, and it cannot be more than 8 characters, or end in a
number;
        fmtname = 'prime';
        hlo = '';
        type = 'C';
```

```
*"label" is the right side of the format, i.e., what is returned;
        label = 'Y';

        output;

*This is the "other" option, what is returned if your key value is not found in the format;

        if (last) then do;
                hlo = 'o';
                label = 'N';
                output;
        end;
run;

*This step actually creates the format;
proc format cntlin=prime;
run;

*The "prime" data set is no longer needed for the format;
proc datasets;
        delete prime;
run;
```

Example 8- Handling PHI Using Random Identifiers in Place of Actual Person Identifiers

```
*
*       PROGRAM NAME  : random.sas
*       DESCRIPTION   : Create a random person ID
*       PROGRAMMER    : Kennell
*;

options nocenter ls=132 ps=32000 obs=max;

libname insidr "/mdr/pub/sidr/fy13/sidr.fy13";
libname out "/hpae2/kennell/training/examples";

*Read in SIDRs for ADSMs with PTSD in the primary diagnosis;
data sidr;
        set insidr.fy13(keep = patuniq combenf dx1);
        where combenf = '4' and dx1 = '30981';
run;

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

*Make random person identifier;
```

```
data people;
	set people;
	random = ranuni(5);
run;

proc sort data = people;
	by random;
run;

*Save a data set with the mapping from EDI_PN (patuniq) to the random person ID (person_id);
data out.people;
	set people;

	person_id = _n_;

	drop random;
run;

proc datasets;
		delete people;
run;

proc sort data = out.people;
		by patuniq;
run;

proc sort data = sidr;
		by patuniq;
run;

*Save PTSD data set with randomized person ID;
data out.pdts;
		merge sidr(in = a) out.people(in = b);
		by patuniq;
		if a and b;

		drop patuniq;
run;
```