Creating and Downloading User-Data Files

Agilent Technologies
Signal Generators

This guide applies to the following signal generator models:

N5181A/82A MXG Signal Generators

E4428C/38C ESG Signal Generators

E8257D/67D PSG Signal Generators

E8663B Analog Signal Generator

Due to our continuing efforts to improve our products through firmware and hardware revisions, signal generator design and operation may vary from descriptions in this guide. We recommend that you use the latest revision of this guide to ensure you have up-to-date product information. Compare the print date of this guide (see bottom of page) with the latest revision, which can be downloaded from the following websites:

http://www.agilent.com/find/psg

http://www.agilent.com/find/mxg

http://www.agilent.com/find/e8663b

http://www.agilent.com/find/esg

Agilent Technologies

Manufacturing Part Number: E4400-90651

Printed in USA

December 2006

Notice

The material contained in this document is provided “as is”, and is subject to being changed, without notice, in future editions.

Further, to the maximum extent permitted by applicable law, Agilent disclaims all warranties, either express or implied with regard to this manual and to any of the Agilent products to which it pertains, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. Agilent shall not be liable for errors or for incidental or consequential damages in connection with the furnishing, use, or performance of this document or any of the Agilent products to which it pertains. Should Agilent have a written contract with the User and should any of the contract terms conflict with these terms, the contract terms shall control.

Creating and Downloading User-Data Files

Contents

Creating and Downloading User-Data Files

NOTE Some features apply to only the E4438C with Option 001, 002, 601, or 602 and E8267D with Option 601 or 602. These exceptions are indicated in the sections.

The following sections and procedures contain remote SCPI commands. For front panel key commands, refer to the *User's Guide*, *Key and Data Field Reference* (ESG, PSG, and E8663B), or to the Key Help in the signal generator.

This information is also available in the signal generator's *Programming Guide*.

This manual explains the requirements and processes for creating and downloading user-data, and contains the following sections:

Overview

User data is a generic term for various data types created by the user and stored in the signal generator. This includes the following data (file) types:

NOTE For the Agilent MXG: Bit, PRAM, FIR Filter, and State data (file) types are not applicable.

Bit	This file type lets the user download payload data for use in streaming or framed signals. It lets the user determine how many bits in the file the signal generator uses.
Binary	This file type provides payload data for use in streaming or framed signals. It differs from the bit file type in that you cannot specify a set number of bits. Instead the signal generator uses all bits in the file for streaming data and all bits that fill a frame for framed data. If there are not enough bits to fill a frame, the signal generator truncates the data and repeats the file from the beginning.
PRAM	With this file type, the user provides the payload data along with the bits to control signal attributes such as bursting. This file type is available for only the real-time Custom and TDMA modulation formats.
FIR Filter	This file type stores user created custom filters.
State	This file type lets the user store signal generator settings, which can be recalled. This provides a quick method for reconfiguring the signal generator when switching between different signal setups.
User Flatness Correction	This file type lets the user store amplitude corrections for frequency points that the signal generator uses during list sweeps.

Prior to creating and downloading files, you need to take into consideration the file size and the amount of remaining signal generator memory. For more information, see "Signal Generator Memory" on page 3

Signal Generator Memory

The signal generator provides two types of memory, volatile and non-volatile.

NOTE User BIT, user FIR folders and User PRAM references are only applicable to the E4438C with Options 001, 002, 601, or 602, and E8267D with Options 601 or 602.

Volatile — Random access memory that does not survive cycling of the signal generator power. This memory is commonly referred to as waveform memory (WFM1) or pattern RAM (PRAM). Refer to Table 1 for the file types that share this memory:

Table 1 Signal Generators and Volatile Memory File Types

Volatile Memory Type	Model of Signal Generator			
	N5182A with Option 651, 652, or 654	E4438C with Option 001, 002, 601, or 602	E8267D Option 601 or 602	All Other models[1]
I/Q	x	x	x	–
Marker	x	x	x	–
File header	x	x	x	–
User PRAM	–	x	x	–
User Binary	x	x	x	–
User Bit	–	x	x	–
Waveform Sequences (multiple I/Q files played together)	x	x	x	–

1. N5181A, E8663B, E4428C, and the E8257D.

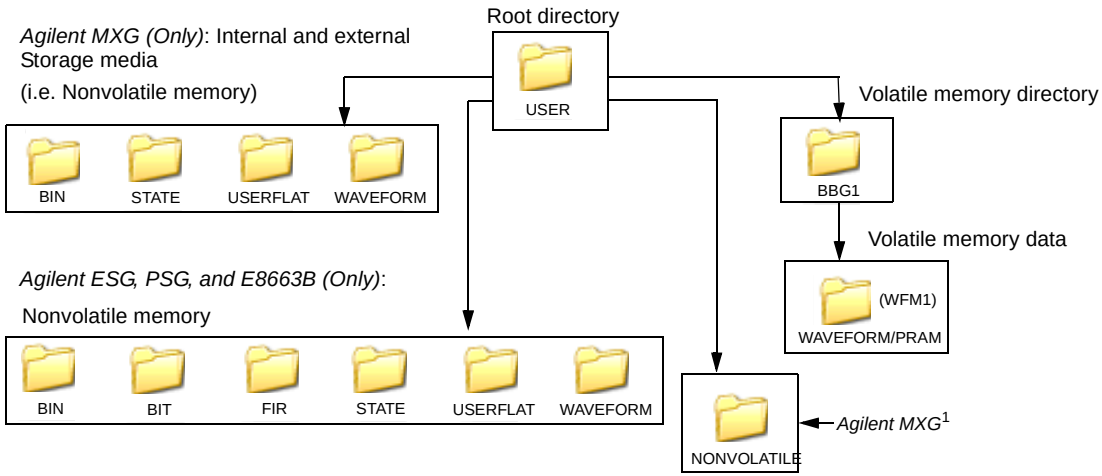
Non-volatile — Storage memory where files survive cycling of the signal generator power. Files remain until overwritten or deleted. Refer to Table 2 on page 4 for the file types that share this memory:

Table 2 Signal Generators and Non-Volatile Memory Types

Non-Volatile Memory Type	Model of Signal Generator			
	N5182A with Option 651, 652, or 654	E4438C with Option 001, 002, 601, or 602	E8267D Option 601 or 602	All Other models[1]
I/Q	x	x	x	x
Marker	x	x	x	x
File header	x	x	x	x
Sweep List	x	x	x	x
User PRAM	–	x	x	–
User Binary	x	x	x	x
User Bit	–	x	x	–
User FIR	–	x	x	–
Instrument State	x	x	x	x
Waveform Sequences (multiple I/Q files played together)	x	x	x	–

1. N5181A, E8663B, E4428C, and the E8257D.

The following figure shows the signal generator's directory structure for the user-data files.



[1]This NONVOLATILE directory shows the files with the same extensions as the USB media and is useful with ftp.

Memory Allocation

Volatile Memory

The signal generator allocates volatile memory in blocks of 1024 bytes. For example, a user-data file with 60 bytes uses 1024 bytes of memory. For a file that is too large to fit into 1024 bytes, the signal generator allocates additional memory in multiples of 1024 bytes. For example, the signal generator allocates 3072 bytes of memory for a file with 2500 bytes.

3 x 1024 bytes = 3072 bytes of memory

As shown in the examples, files can cause the signal generator to allocate more memory than what is actually used, which decreases the amount of available memory.

User-data blocks consist of 1024 bytes of memory. Each user-data file has a file header that uses 512 bytes for the Agilent MXG, or 256 bytes for the ESG/PSG in the first data block for each user-data file.

Non-Volatile Memory (Agilent MXG)

On the N5182A, non-volatile files are stored on the non-volatile internal signal generator memory (i.e. internal storage) or to the USB media, if available. The Agilent MXG non-volatile internal memory allocated according to a Microsoft compatible file allocation table (FAT) file system. The Agilent MXG signal generator allocates non-volatile memory in clusters according to the drive size (see table on page 7). For example, referring to table on page 7, if the drive size is 15 MB and if the file is less than or equal to 4k bytes, the file uses only one 4 KB cluster of memory. For files larger than 4 KB, and with a drive size of 15 MB, the signal generator allocates additional memory in multiples of 4KB clusters. For example, a file that has 21,538 bytes consumes 6 memory clusters (24,000 bytes).

On the Agilent MXG the non-volatile memory is *also* referred to as internal storage and USB media. The Internal and USB media files /USERS/NONVOLATILE Directory contains file names with full extensions (i.e. .marker, .header, etc.-).

For more information on default cluster sizes for FAT file structures, refer to Table 3 on page 7 and to *http://support.microsoft.com/*.

Table 3

Drive Size (logical volume)	Cluster Size (Bytes) (Minimum Allocation Size)
O MB - 15 MB	4K
16 MB - 127 MB	2K
128 MB - 255 MB	4K
256 MB - 511 MB	8K
512 MB - 1023 MB	16k
1024 MB - 2048 MB	32K
2048 MB - 4096 MB	64K
4096 MB - 8192 MB	128K
8192 MB - 16384 MB	256K

Non-Volatile Memory (ESG, PSG, and E8663B)

The signal generator allocates non-volatile memory in blocks of 512 bytes. For files less than or equal to 512 bytes, the file uses only one block of memory. For files larger than 512 bytes, the signal generator allocates additional memory in multiples of 512 byte blocks. For example, a file that has 21,538 bytes consumes 43 memory blocks (22,016 bytes).

Memory Size

For the E4438C, E8267D, and E8663B, the maximum volatile memory size for user data is less than the maximum size for waveform files. This is because the signal generator permanently allocates a portion of the volatile memory for waveform markers. The values in Table 4 is the total amount of memory after deducting the waveform marker memory allocation.

The amount of available memory, volatile and non-volatile, varies by signal generator option and the size of the other files that share the memory. The baseband generator (BBG) options contain the volatile memory. Table 4 shows the maximum available memory assuming that there are no other files residing in memory.

Table 4 Maximum Signal Generator Memory

Volatile (WFM1/PRAM) Memory			Non-Volatile (NVWFM) Memory	
Option	**Size**		**Option**	**Size**
N5182A				
651, 652, 654 (BBG)	40 MB		**Standard (N5181A)**	8 MB
019	320 MB		**Standard (N5182A)**	512 MB
----	----		**USB memory stick**	*user determined*
E4438C and E8267D				
001, 601 (BBG)[1,2]	32 MB		**Standard**	512 MB
002 (BBG)[1,2]	128 MB		**005 (Hard disk)**[2]	6 GB
602 (BBG)[2]	256 MB		----	----

1. Options 001 and 002 apply to only the E4438C ESG.
2. Not available on the E8663B.

Checking Available Memory

Whenever you download a user-data file, you must be aware of the amount of remaining signal generator memory. Table 5 shows to where each user-data file type is downloaded and from which memory type the signal generator accesses the file data. Information on downloading a user-data file is located within each user-data file section.

NOTE The Bit, PRAM, FIR filter, and State user-data (file) types only apply to the E4438C with Option 001, 002, 601, or 602, and the E8267D with Option 601 or 602.

Table 5 User-Data File Memory Location

User-Data File Type	Download Memory	Access Memory
Bit	Non-volatile	Volatile
Binary	Non-volatile	Volatile
PRAM	Volatile	Volatile

Table 5 User-Data File Memory Location

User-Data File Type	Download Memory	Access Memory
Instrument State	Non-volatile	Non-volatile
FIR	Non-volatile	Non-volatile
Flatness	Non-volatile	Non-volatile

Bit and binary files increase in size when the signal generator loads the data from non-volatile to volatile memory. For more information, see "User File Size" on page 15.

Use the following SCPI commands to determine the amount of remaining memory:

Volatile Memory `:MMEM:CAT? "WFM1"`

The query returns the following information:

`<memory used>,<memory remaining>,<"file_names">`

Non-Volatile Memory `:MEM:CAT:ALL?`

The query returns the following information:

`<memory used>,<memory remaining>,<"file_names">`

NOTE The signal generator calculates the memory values based on the number of bytes used by the files residing in volatile or non-volatile memory, and not on the memory block allocation. To accurately determine the available memory, you must calculate the number of blocks of memory used by the files. For more information on memory block allocation, see "Memory Allocation" on page 5.

User File Data (Bit / Binary) Downloads (E4438C and E8267D)

NOTE This section applies only to the E4438C with Option 001, 002, 601, or 602, and the E8267D with Option 601 or 602.

If you encounter problems with this section, refer to "Data Transfer Troubleshooting (E4438C and E8267D Only)" on page 64.

The signal generator accepts externally created and downloaded user file data for real-time modulation formats that have user file as a data selection (shown as <"file_name"> in the data selection SCPI command). When you select a user file, the signal generator incorporates the user file data (payload data) into the modulation format's data fields. You can create the data using programs such as MATLAB or Mathcad. The following table shows the available real-time modulation formats by signal generator model:

E4438C ESG		E2867D PSG
CDMA[1]	TDMA[2]	Custom[c]
Custom[3]	W-CDMA[4]	
GPS[5]	---	

1. Requires Option 401.
2. Real-time TDMA modulation formats require Option 402 and include EDGE, GSM, NADC, PDC, PHS, DECT, and TETRA.
3. For ESG, requires Option 001, 002, 601, or 602, for PSG requires Option 601 or 602.
4. Requires Option 400.
5. Requires Option 409.

The signal generator uses two file types for downloaded user file data: bit and binary. With a bit file, the signal generator views the data up to the number of bits specified when the file was downloaded. For example, if you specify to use 153 bits from a 160 bit (20 bytes) file, the signal generator transmits 153 bits and ignores the remaining 7 bits. This provides a flexible means in which to control the number of transmitted data bits. It is the preferred file type and the easiest one to use.

With a binary file, the signal generator sees all bytes (bits) in a downloaded file and attempts to use them. This can present challenges especially when working with framed data. In this situation, your file needs to contain enough bits to fill a frame or timeslot, or multiple frames or timeslots, to end on the desired boundary. To accomplish this, you may have to remove or add bytes. If there are not enough bits remaining in the file to fill a frame or timeslot, the signal generator truncates the data causing a discontinuity in the data pattern.

You download a user file to either the Bit or Binary memory catalog (directory). Unlike a PRAM file (covered later in this chapter), user file data does not contain control bits, it is just data. The signal generator adds control bits to the user file data when it generates the signal. There are two ways that the signal generator uses the data, either in a continuous data pattern (unframed) or within

framed boundaries. Real-time Custom uses only unframed data, real-time TDMA modulation formats use both types, and the others use only framed data.

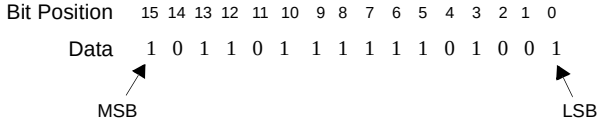
NOTE For unframed data transmission, the signal generator requires a minimum of 60 symbols. For more information, see "Determining Memory Usage for Custom and TDMA User File Data" on page 16.

You create the user file to either fill a single timeslot/frame or multiple timeslots/frames. To create multiple timeslots/frames, simply size the file with enough data to fill the number of desired timeslots/frames

User File Bit Order (LSB and MSB)

The signal generator views the data from the most significant bit (MSB) to the least significant bit (LSB). When you create your user file data, it is important that you organize the data in this manner. Within groups (strings) of bits, a bit's value (significance) is determined by its location in the string. The following shows an example of this order using two bytes.

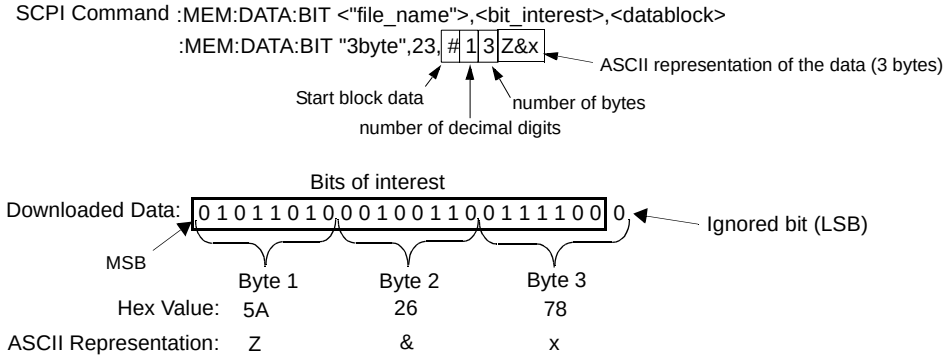
Most Significant Bit (MSB)	This bit has the highest value (greatest weight) and is located at the far left of the bit string.
Least Significant Bit (LSB)	This bit has the lowest value (bit position zero) and is located at the far right of the bit string.



Bit File Type Data

The bit file is the preferred file type and the easiest to use. When you download a bit file, you designate how many bits in the file the signal generator can modulate onto the signal. During the file download, the signal generator adds a 10-byte file header that contains the information on the number of bits the signal generator is to use.

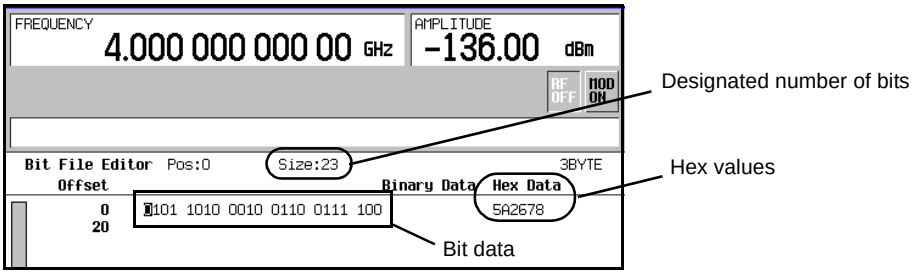
Although you download the data in bytes, when the signal generator uses the data, it recognizes only the bits of interest that you designate in the SCPI command and ignores the remaining bits. This provides greater flexibility in designing a data pattern without the concern of using an even number of bytes as is needed with the binary file data format. The following figure illustrates this concept. The example in the figure shows the bit data SCPI command formatted to download three bytes of data, but only 23 bits of the three bytes are designated as the bits of interest. (For more information on the bit data SCPI command format, see "Downloading User Files" on page 19 and "Command for Bit File Downloads" on page 22.)



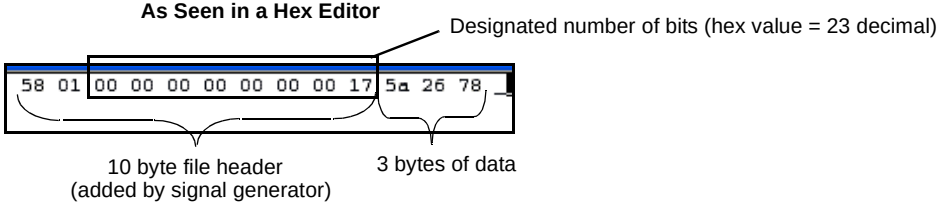
The following figure shows the same downloaded data from the above example as viewed in the signal generator's bit file editor (see the *User's Guide* for more information) and with using an external hex editor program.

SCPI command to download the data :MEM:DATA:BIT "3byte",23,#13Z&x

As Seen in the Signal Generator's Bit File Editor



As Seen in a Hex Editor

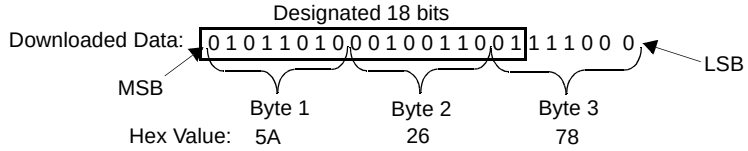


In the bit editor, notice that the ignored bit of the bit-data is not displayed, however the hex value still shows all three bytes. This is because bits 1 through 7 are part of the first byte, which is shown as ASCII character x in the SCPI command line. The view from the hex editor program confirms that the downloaded three bytes of data remains unchanged. To view a downloaded bit file with an external hex editor program, FTP the file to your PC/UNIX workstation. For information on how to FTP a file, see "FTP Procedures" on page 26.

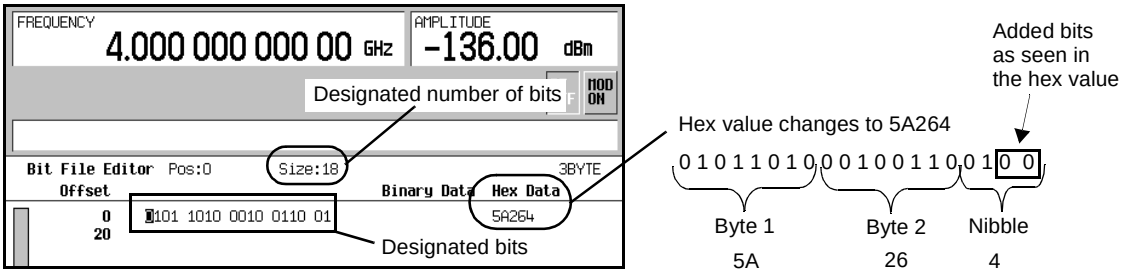
Even though the signal generator views the downloaded data on a bit basis, it groups the data into bytes, and when the designated number of bits is not a multiple of 8 bits, the last byte into one or

more 4-bit nibbles. To make the last nibble, the signal generator adds bits with a value of zero. The signal generator does not show the added bits in the bit editor and ignores the added bits when it modulates the data onto the signal, but these added bits do appear in the hex value displayed in the bit file editor. The following example, which uses the same three bytes of data, further demonstrates how the signal generator displays the data when only two bits of the last byte are part of the bits of interest.

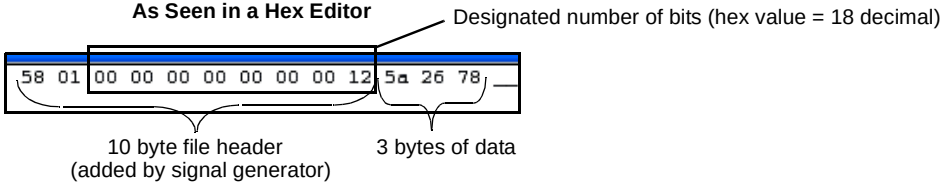
SCPI command to download the data :MEM:DATA:BIT "3byte",18,#13Z&x



As Seen in the Signal Generator's Bit File Editor



As Seen in a Hex Editor



Notice that the bit file editor shows only two bytes and one nibble. In addition, the signal generator shows the nibble as hex value 4 instead of 7 (78 is byte 3—ASCII character x in the SCPI command line). This is because the signal generator sees bits 17 and 18, and assumes bits 19 and 20 are 00. As viewed by the signal generator, this makes the nibble 0100. Even though the signal generator extrapolates bits 19 and 20 to complete the nibble, it ignores these bits along with bits 21 through 24. As seen with the hex editor program, the signal generator does not actually change the three bytes of data in the downloaded file.

For information on editing a file after downloading, see "Modifying User File Data" on page 25.

Binary File Type Data

With the Binary file type, the signal generator sees all of the bytes within the downloaded file and attempts to use all of the data bits. When using this file type, the biggest challenge is creating the data, so that the signal generator uses all of the bits (bytes) contained within the file. This is referred to as using an even number of bytes. The method of creating the user file data pattern depends on whether you are using unframed or framed data. The following two sections illustrate the complexities of using the binary file format. You can eliminate these complexities by using the bit file format (see "Bit File Type Data" on page 11).

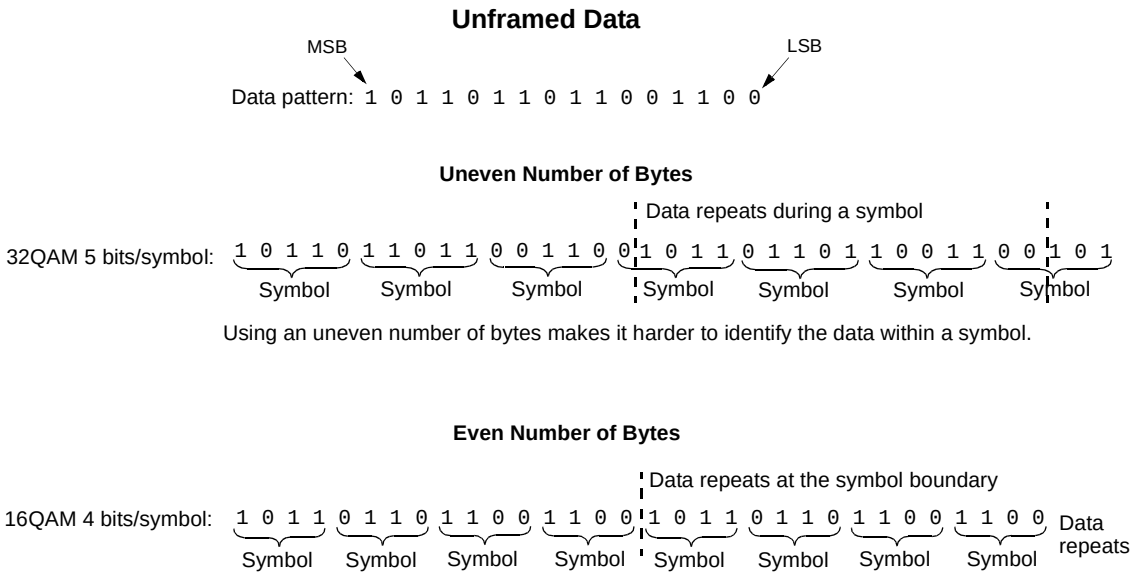
Unframed Binary Data

When creating unframed data, you must think in terms of bits per symbol; so that your data pattern begins and ends on the symbol boundary, with an even number of bytes. For example, to use 16QAM modulation, the user file needs to contain 32 bytes:

- enough data to fill 16 states 4 times
- end on a symbol boundary
- create 64 symbols (the signal generator requires a minimum of 60 symbols for unframed data)

To do the same with 32QAM, requires a user file with 40 bytes.

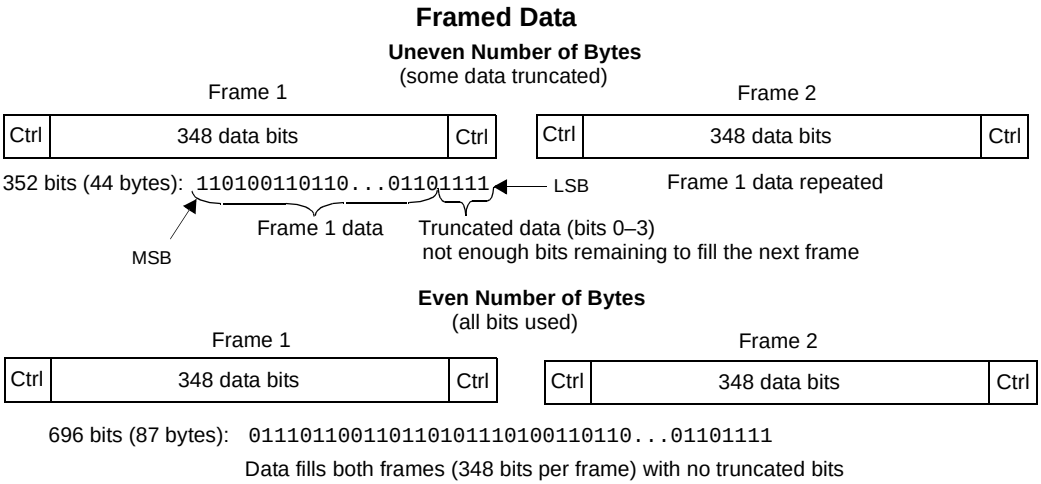
When you do *not* use an even number of bytes, the signal generator repeats the data in the same symbol where the data stream ends. This means that your data would not end on the symbol boundary, but during a symbol. This makes it harder to identify the data content of a symbol. The following figure illustrates the use of an uneven number of bytes and an even number of bytes.



Framed Binary Data

When using framed data, ensure that you use an even number of bytes and that the bytes contain enough bits to fill the data fields within a timeslot or frame. When there are not enough bits to fill a single timeslot or frame, the signal generator replicates the data pattern until it fills the timeslot/frame.

The signal generator creates successive timeslots/frames when the user file contains more bits than what it takes to fill a single timeslot or frame. When there are not enough bits to completely fill successive timeslots or frames, the signal generator truncates the data at the bit location where there is not enough bits remaining and repeats the data pattern. This results in a data pattern discontinuity. For example, a frame structure that uses 348 data bits requires a minimum file size of 44 bytes (352 bits), but uses only 43.5 bytes (348 bits). In this situation, the signal generator truncates the data from bit 3 to bit 0 (bits in the last byte). Remember that the signal generator views the data from MSB to LSB. For this example to have an even number of bytes and enough bits to fill the data fields, the file needs 87 bytes (696 bits). This is enough data to fill two frames while maintaining the integrity of the data pattern, as illustrated in the following figure.



For information on editing a file after downloading, see "Modifying User File Data" on page 25.

User File Size

You download user files into non-volatile memory. For CDMA, GPS, and W-CDMA, the signal generator accesses the data directly from non-volatile memory, so the file size up to the maximum file size (shown in Table 6) for these formats is limited only by the amount of available non-volatile memory. As seen in the table, the baseband generator option does not affect these file sizes.

For Custom and TDMA, however, when the signal generator creates the signal, it loads the data from non-volatile memory into volatile memory, which is also the same memory that the signal generator uses for Arb-based waveforms. For user data files, volatile memory is commonly referred to as pattern ram memory (PRAM). Because the Custom and TDMA user files use volatile memory, their maximum file size depends on the baseband generator (BBG) option and the amount of available

PRAM. (Volatile memory resides on the BBG.) Table 6 shows the maximum file sizes by modulation format and baseband generator option.

Table 6 Maximum User File Size

Modulation Format	Baseband Generator Option		
	001, 601	002	602
Custom[1] TDMA[a]	800 kB	3.2 MB	6.4 MB
CDMA[2] GPS[b] W-CDMA[b]	10 kB	10 kB	10 kB

1. File size with no other files residing in volatile memory.
2. File size is not affected by the BBG option.

For more information on signal generator memory, see "Signal Generator Memory" on page 3. To determine how much memory is remaining in non-volatile and volatile memory, see "Checking Available Memory" on page 8.

Determining Memory Usage for Custom and TDMA User File Data

For Custom and TDMA user files, the signal generator uses both non-volatile and volatile (PRAM/waveform) memory: you download the user file to non-volatile memory. To determine if there is enough non-volatile memory, check the available non-volatile memory and compare it to the size of the file to be downloaded.

After you select a user file and turn the format on, the signal generator loads the file into volatile memory for processing:

- It translates each data bit into a 32-bit word (4 bytes).

  The 32-bit words are not saved to the original file that resides in non-volatile memory.
- It creates an expanded data file named AUTOGEN_PRAM_1 in volatile memory while also maintaining a copy of the original file in volatile memory. It is the AUTOGEN_PRAM_1 file that contains the 32-bit words and accounts for most of the user file PRAM memory space.
- If the transmission is using unframed data and there are not enough bits in the data file to create 60 symbols, the signal generator replicates the data pattern until there is enough data for 60 symbols. For example, GSM uses 1 bit per symbol. If the user file contains only 24 bits, enough for 24 symbols, the signal generator replicates the data pattern two more times to create a file with 72 bits. The expanded AUTOGEN_PRAM_1 file size would show 288 bytes (72 bits $\times$ 4 bytes/bit).

Use the following procedures to calculate the required amount of volatile memory for both framed and unframed TDMA signals:

- "Calculating Volatile Memory (PRAM) Usage for Unframed Data" on page 17
- "Calculating Volatile Memory (PRAM) Usage for Framed Data" on page 17

Calculating Volatile Memory (PRAM) Usage for Unframed Data

Use this procedure to calculate the memory size for either a bit or binary file. To properly demonstrate this process, the procedure employs a user file that contains 70 bytes (560 bits), with the bit file using only 557 bits.

1. Determine the AUTOGEN_PRAM_1 file size:

   The signal generator creates a 32-bit word for each user file bit (1 bit equals 4 bytes).

   Binary file 4 bytes × (70 bytes x 8 bits) = 2240 bytes

   Bit file 4 bytes × 557 bits= 2228 bytes

2. Calculate the number of memory blocks that the AUTOGEN_PRAM_1 file will occupy:

   Volatile memory allocates memory in blocks of 1024 bytes.

   Binary file 2240 / 1024 = 2.188 blocks

   Bit file 2228 / 1024 = 2.176 blocks

3. Round the memory block value to the next highest integer value.

   For this example, the AUTOGEN_PRAM_1 file will use three blocks of memory for a total of 3072 bytes.

4. Determine the number of memory blocks that the copy of the original file occupies in volatile memory.

   For this example the bit and binary file sizes are shown in the following list:

   - Binary file = 70 bytes < 1024 bytes = 1 memory block
   - Bit file = 80 bytes < 1024 bytes = 1 memory block

     Remember that a bit file includes a 10-byte file header.

5. Calculate the total volatile memory occupied by the user file data:

AUTOGEN_PRAM_1	Original File
3 blocks	1 block
1024 (3 + 1) = 4096 bytes	

Calculating Volatile Memory (PRAM) Usage for Framed Data

Framed data is not a selection for Custom, but it is for TDMA formats. To frame data, the signal generator adds framing overhead data such as tail bits, guard bits, and sync bits. These framing bits are in addition to the user file data. For more information on framed data, see "Understanding Framed Transmission For Real-Time TDMA" on page 27.

When using framed data, the signal generator views the data (framing and user file bits) in terms of the number of bits per frame, even if only one timeslot within a frame is active. This means that the signal generator creates a 32-bit word for each bit in a frame, for both active and inactive timeslots.

You can create a user file so that it fills a timeslot once or multiple times. When the user file fills a timeslot multiple times, the signal generator creates the same number of frames as the number of timeslots that the user file fills. For example, if a file contains enough data to fill a timeslot three times, the signal produces three new frames before the frames repeat. Each new frame increases the

AUTOGEN_PRAM_1 file size. If you select different user files for the timeslots within a frame, the user file that produces the largest number of frames determines the size of the AUTOGEN_PRAM_1 file.

Use this procedure to calculate the volatile memory usage for a GSM signal with two active timeslots and two user binary files. One user file, 57 bytes, is for a normal timeslot and another, 37 bytes, is for a custom timeslot.

1. Determine the total number of bits per timeslot.

   A GSM timeslot consists of 156.25 bits (control and payload data).

2. Calculate the number of bits per frame.

   A GSM frame consists of 8 timeslots: 8 × 156.25 = 1250 bits per frame

3. Determine how many bytes it takes to produce one frame in the signal generator:

   The signal generator creates a 32-bit word for each bit in the frame (1 bit equals 4 bytes).

   4 x 1250 = 5000 bytes

   Each GSM frame uses 5000 bytes of PRAM memory.

4. Analyze how many timeslots the user file data will fill.

   A normal GSM timeslot (TS) uses 114 payload data bits, and a custom timeslot uses 148 payload data bits. The user file (payload data) for the normal timeslot contains 57 bytes (456 bits) and the user file for the custom timeslot contains 37 bytes (296 bits).

   Normal TS 456 / 114 = 4 timeslots

   Custom TS 296 / 148 = 2 timeslots

NOTE Because there is an even number of bytes, either a bit or binary file works in this scenario. If there was an uneven number of bytes, a bit file would be the best choice to avoid data discontinuity.

---

5. Compute the number of frames that the signal generator will generate.

   There is enough user file data for four normal timeslots and two custom timeslots, so the signal generator will generate four frames of data.

6. Calculate the AUTOGEN_PRAM_1 file size:

Number of Frames	Bytes per Frame
4	5000
4 x 5000 = 20000 bytes	

7. Calculate the number of memory blocks that the AUTOGEN_PRAM_1 file will occupy:

   Volatile memory allocates memory in blocks of 1024 bytes.

   20000 / 1024 = 19.5 blocks

8. Round the memory block value up to the next highest integer value.

   For this example, the AUTOGEN_PRAM_1 file will use 20 blocks of memory for a total of 20480 bytes.

9. Determine the number of memory blocks that the original files occupy in volatile memory.

   The files do not share memory blocks, so you must determine how many memory blocks each file occupies.

Normal TS	Custom TS
57 bytes = 1 block	37 bytes = 1 block
1 + 1 = 2 memory blocks	

NOTE If the user file type is bit, remember to include the 10-byte file header in the file size.

10. Calculate the total volatile memory occupied by the AUTOGEN_PRAM_1 file and the user files:

AUTOGEN_PRAM_1	User Files
20 blocks	2 blocks
1024 (20 + 2) = 22528 bytes	

Downloading User Files

The signal generator expects bit and binary file type data to be downloaded as block data (binary data in bytes). The IEEE standard 488.2-1992 section 7.7.6 defines block data.

This section contains two examples to explain how to format the SCPI command for downloading user file data. The examples use the binary user file SCPI command, however the concept is the same for the bit file SCPI command:

- Command Format
- "Command Format in a Program Routine" on page 20

Command Format

This example conceptually describes how to format a data download command (#ABC represents the block data):

```
:MEM:DATA <"file_name">,#ABC
```

<"file_name">	the data file path and name
#	indicates the start of the block data
A	the number of decimal digits present in B
B	a decimal number specifying the number of data bytes to follow in C

C	the file data in bytes

```
:MEM:DATA "bin:my_file",#3240 12%S!4&07#8g*Y9@7...
```

file location file_name A B C

bin:	the location of the file within the signal generator file system
my_file	the data file name as it will appear in the signal generator's memory catalog
#	indicates the start of the block data
3	B has three decimal digits
240	240 bytes (1,920 bits) of data to follow in C
12%S!4&07#8g*Y9@7...	the ASCII representation of some of the block data (binary data) downloaded to the signal generator, however not all ASCII values are printable

In actual use, the block data is not part of the command line as shown above, but instead resides in a binary file on the PC/UNIX. When the program executes the SCPI command, the command line notifies the signal generator that it is going to receive block data of the stated size and to place the file in the signal generator file directory with the indicated name. Immediately following the command execution, the program downloads the binary file to the signal generator. This is shown in the following section, "Command Format in a Program Routine"

Some commands are file location specific and do not require the file location as part of the file name. An example of this is the bit file SCPI command shown in "Command for Bit File Downloads" on page 22.

Command Format in a Program Routine

This section demonstrates the use of the download SCPI command within the confines of a C++ program routine. The following code sends the SCPI command and downloads user file data to the signal generator's Binary memory catalog (directory).

Line **Code—Download User File Data**

```
1  int bytesToSend;
2  bytesToSend = numsamples;
3  char s[20];
4  char cmd[200];
5  sprintf(s, "%d", bytesToSend);
6  sprintf(cmd, ":MEM:DATA \"BIN:FILE1\", #%d%d", strlen(s), bytesToSend);
7  iwrite(id, cmd, strlen(cmd), 0, 0);
8  iwrite(id, databuffer, bytesToSend, 0, 0);
9  iwrite(id, "\n", 1, 1, 0);
```

Line	Code Description—Download User File Data
1	Define an integer variable (*bytesToSend*) to store the number of bytes to send to the signal generator.
2	Calculate the total number of bytes, and store the value in the integer variable defined in line 1.
3	Create a string large enough to hold the *bytesToSend* value as characters. In this code, string *s* is set to 20 bytes (20 characters—one character equals one byte)
4	Create a string and set its length (*cmd*[200]) to hold the SCPI command syntax and parameters. In this code, we define the string length as 200 bytes (200 characters).
5	Store the value of *bytesToSend* in string *s*. For example, if bytesToSend = 2000; *s* = ”2000”. *sprintf()* is a standard function in C++, which writes string data to a string variable.
6	Store the SCPI command syntax and parameters in the string *cmd*. The SCPI command prepares the signal generator to accept the data. • *strlen()* is a standard function in C++, which returns length of a string. • If *bytesToSend* = 2000, then *s* = “2000”, *strlen(s)* = 4, so *cmd* = :MEM:DATA ”BIN:FILE1\” #42000.
7	Send the SCPI command stored in the string *cmd* to the signal generator contained in the variable *id*. • *iwrite()* is a SICL function in Agilent IO library, which writes the data (block data) specified in the string *cmd* to the signal generator. • The third argument of *iwrite()*, *strlen(cmd)*, informs the signal generator of the number of bytes in the command string. The signal generator parses the string to determine the number of data bytes it expects to receive. • The fourth argument of iwrite(), 0, means there is no END of file indicator for the string. This lets the session remain open, so the program can download the user file data.

Line	Code Description—Download User File Data
8	Send the user file data stored in the array *(databuffer)* to the signal generator. • *iwrite()* sends the data specified in *databuffer* to the signal generator (session identifier specified in *id*). • The third argument of *iwrite()*, *bytesToSend*, contains the length of the *databuffer* in bytes. In this example, it is 2000. • The fourth argument of *iwrite()*, 0, means there is no END of file indicator in the data. In many programming languages, there are two methods to send SCPI commands and data: — Method 1 where the program stops the data download when it encounters the first zero (END indicator) in the data. — Method 2 where the program sends a fixed number of bytes and ignores any zeros in the data. This is the method used in our program. For your programming language, you must find and use the equivalent of method two. Otherwise you may only achieve a partial download of the user file data.
9	Send the terminating carriage (\n) as the last byte of the waveform data. • *iwrite()* writes the data "\n" to the signal generator (session identifier specified in *id).* • The third argument of *iwrite()*, 1, sends one byte to the signal generator. • The fourth argument of *iwrite(),* 1, is the END of file indicator, which the program uses to terminate the data download. To verify the user file data download, see "Command for Bit File Downloads" on page 22 and "Commands for Binary File Downloads" on page 23.

Command for Bit File Downloads

Because the signal generator adds a 10-byte file header during a bit file download, you must use the SCPI command shown in Table 7. If you FTP or copy the file for the initial download, the signal generator does not add the 10-byte file header, and it does recognize the data in the file (no data in the transmitted signal).

Bit files enable you to control how many bits in the file the signal generator modulates onto the signal. Even with this file type, the signal generator requires that all data be contained within bytes. For more information on bit files, see "Bit File Type Data" on page 11.

Table 7 Bit File Type SCPI Commands

Type	Command Syntax
Command	`:MEM:DATA:BIT <"file_name">,<bit_count>,<block_data>` This downloads the file to the signal generator.

Table 7 Bit File Type SCPI Commands

Type	Command Syntax
Query	:MEM:DATA:BIT? <"file_name"> Within the context of a program this query extracts the user file data. Executing the query in a command window causes it to return the following information: <bit_count>,<block_data>.
Query	:MEM:CAT:BIT? This lists all of the files in the bit file directory and shows the remaining non-volatile memory: <bytes used by bit files>,<available non-volatile memory>,<"file_names">

Command Syntax Example

The following command downloads a file that contains 17 bytes:

```
:MEM:DATA:BIT "new_file",131,#21702%S!4&07#8g*Y9@7
```

Since this command is file specific (BIT), there is no need to add the file path to the file name.

After execution of this command, the signal generator creates a file in the bit directory (memory catalog) named “new_file” that contains 27 bytes. Remember that the signal generator adds a 10-byte file header to a bit file. When the signal generator uses this file, it will recognize only 131 of the 136 bits (17 bytes) contained in the file.

For information on downloading block data, see “Downloading User Files” on page 19.

Commands for Binary File Downloads

To download a user file as a binary file type means that the signal generator, when the file is selected for use, sees all of the data contained within the file. For more information on binary files, see “Binary File Type Data” on page 14. There are two ways to download the file: to be able to extract the file or not. Each method uses a different SCPI command, which is shown in Table 8.

Table 8 Binary File Type Commands

Command Type		Command Syntax
For Extraction	SCPI	:MEMory:DATA:UNPRotected "bin:file_name",<datablock> This downloads the file to the signal generator. You can extract the file within the context of a program.
	FTP[1]	put <file_name> /user/bin/file_name
No extraction		:MEM:DATA "bin:file_name",<block data> This downloads the file to the signal generator. You cannot extract the file.

Table 8 Binary File Type Commands

Command Type		Command Syntax
Query		:MEM:DATA? "bin:file_name" This returns information on the named file: <bit_count>,<block_data>. Within the context of a program, this query extracts the user file, provided it was download with the proper command.
Query		:MEM:CAT:BIN? This lists all of the files in the bit file directory and shows the remaining non-volatile memory: <bytes used by bit files>,<available non-volatile memory>,<"file_names">

1. See "FTP Procedures" on page 26.

File Name Syntax

There are three ways to format the file name, which must also include the file path:

- "BIN:file_name"
- "file_name@BIN"
- "/user/BIN/file_name"

Command Syntax Example

The following command downloads a file that contains 34 bytes:

```
:MEM:DATA "BIN:new_file",#2347^%S!4&07#8g*Y9@7.?:*Ru[+@y3#_^,>l
```

After execution of this command, the signal generator creates a file in the Binary (Bin) directory (memory catalog) named "new_file" that contains 34 bytes.

For information on downloading block data, see "Downloading User Files" on page 19.

Selecting a Downloaded User File as the Data Source

This section describes how to format SCPI commands for selecting a user file using commands from the GSM and Custom modulation formats. While the commands shown come from only two formats, the concept remains the same when making the data selection for any of the other real-time modulation formats that accept user data. To find the data selection commands for both framed and unframed data for the different modulation formats, see the signal generator's *SCPI Command Reference*.

1. For TDMA formats, select either framed or unframed data:

   ```
   :RADio:GSM:BURSt ON|OFF|1|0
   ```

   ON(1) = framed OFF(0) = unframed

2. Select the user file:

   Unframed Data

```
:RADio:CUSTom:DATA "BIT:file_name"

:RADio:CUSTom:DATA "BIN:file_name"
```

Framed Data

```
:RADio:GSM:SLOT0|1|2|3|4|5|6|7:NORMal:ENCRyption "BIT:file_name"

:RADio:GSM:SLOT0|1|2|3|4|5|6|7:NORMal:ENCRyption "BIN:file_name"
```

3. Configure the remaining signal parameters.
4. Turn the modulation format on:

   ```
   :RADio:CUSTom:STATe On
   ```

Modulating and Activating the Carrier

Use the following commands to modulate the carrier and turn on the RF output. For a complete listing of SPCI commands, refer to the *SCPI Command Reference*.

```
:FREQuency:FIXed 2.5GHZ

:POWer:LEVel -10.0DBM

:OUTPut:MODulation:STATe ON

:OUTPut:STATe ON
```

Modifying User File Data

There are two ways to modify a file after downloading it to the signal generator:

- Use the signal generator's bit file editor. This works for both bit and binary files, but it converts a binary file to a bit file and adds a 10-byte file header. For more information on using the bit file editor, see the signal generator's *User's Guide*. You can also access the bit editor remotely using the signal generator's web server. For web server information, see the signal generator's *Programming Guide*.
- Use a hex editor program on your PC or UNIX workstation, as described below.

Modifying a Binary File with a Hex Editor

1. FTP the file to your PC/UNIX.

   For information on using FTP, see FTP Procedures. Ensure that you use binary file transfers during FTP operations.
2. Modify the file using a hex editor program.
3. FTP the file to the signal generator's BIN memory catalog (directory).

Modifying a Bit File with a Hex Editor

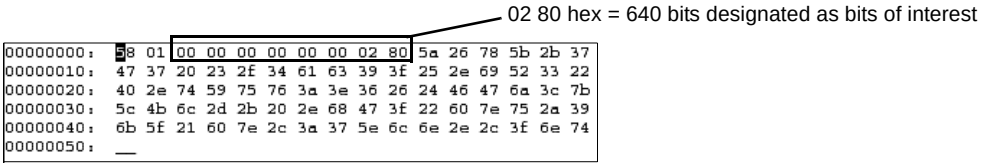
1. FTP the file to your PC/UNIX.

   For information on using FTP, see FTP Procedures. Ensure that you use binary file transfers during FTP operations.

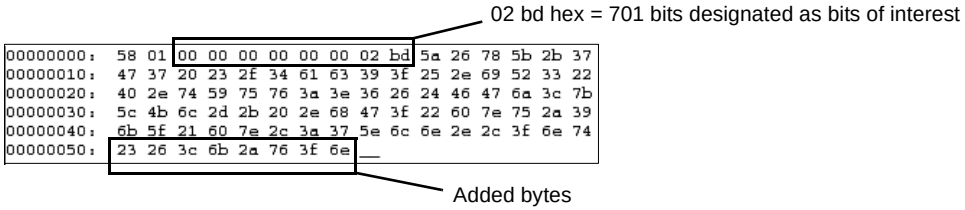
2. Modify the file using a hex editor program.

   If you need to decrease or increase the number of bits of interest, change the file header hex value.

80 Byte File From Signal Generator



Modified File (80 Bytes to 88 Bytes)



3. FTP the file to the signal generator's BIT memory catalog (directory).

FTP Procedures

There are three ways to FTP a file:

- use Microsoft's ® Internet Explorer FTP feature
- use the signal generator's internal web server (ESG firmware ≥ C.03.76)
- use the PC or UNIX command window

Using Microsoft's Internet Explorer

1. Enter the signal generator's hostname or IP address as part of the FTP URL.

   ftp://<host name> or <IP address>

2. Press **Enter** on the keyboard or **Go** from the Internet Explorer window.

   The signal generator files appear in the Internet Explorer window.

3. Drag and drop files between the PC and the Internet Explorer window

---

Microsoft is a U.S registered trademark of Microsoft Corporation.

Using the Signal Generator's Internal Web Server

1. Enter the signal generator's hostname or IP address in the URL.

   http://<host name> or <IP address>

2. Click the **Signal Generator FTP Access** button located on the left side of the window.

   The signal generator files appear in the web browser's window.

3. Drag and drop files between the PC and the browser's window

For more information on the web server feature, see the *Programming Guide*.

Using the Command Window (PC or UNIX)

1. From the PC command prompt or UNIX command line, change to the proper directory:
   - When downloading from the signal generator, the directory in which to place the file.
   - When downloading to the signal generator, the directory that contains the file.
2. From the PC command prompt or UNIX command line, type `ftp <instrument name>`.

   Where `instrument name` is the signal generator's hostname or IP address.
3. At the `User:` prompt, press **Enter** (no entry is required).
4. At the `Password:` prompt, press **Enter** (no entry is required).
5. At the `ftp` prompt, type the desired command:

   To Get a File From the Signal Generator

   ```
   get /user/<directory>/<file_name1> <file_name>
   ```

   To Place a File in the Signal Generator

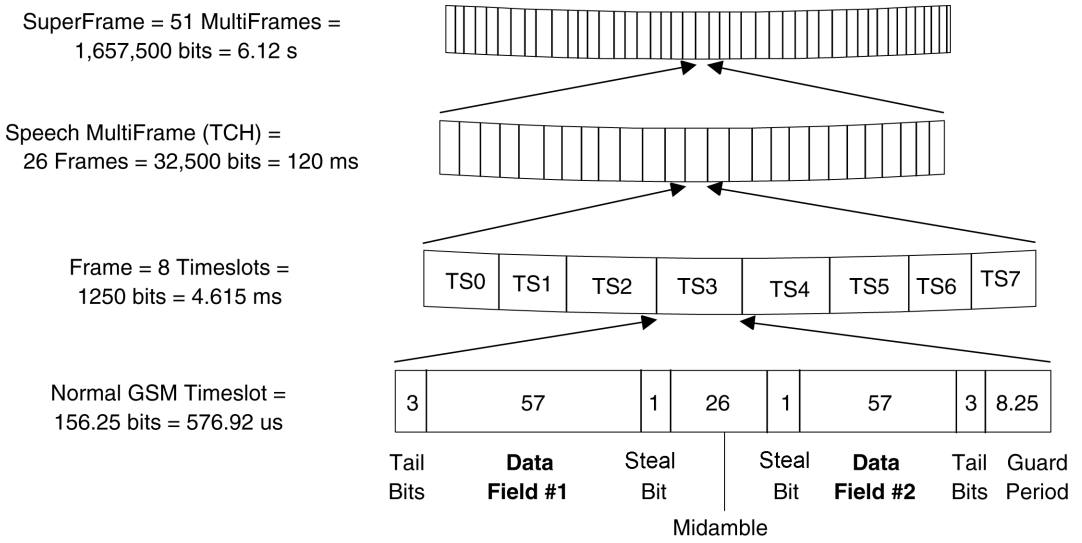
   ```
   put <file_name> /user/<directory>/<file_name1>
   ```

   - `<file_name1>` is the name of the file as it appears in the signal generator's directory.
   - `<file_name>` is the name of the file as it appears in the PC/UNIX current directory.
   - `<directory>` is the signal generator's `BIT` or `BIN` directory.
6. At the `ftp` prompt, type: `bye`
7. At the command prompt, type: `exit`

Understanding Framed Transmission For Real-Time TDMA

Specifying a user file as the data source for a framed transmission provides you with an easy method to multiplex real data into internally generated TDMA framing. The user file fills the data fields of the active timeslot in the first frame, and continue to fill the same timeslot of successive frames as long as there is more data in the file with enough bits to fill the data field. This functionality enables a communications system designer to download and modulate proprietary data sequences, specific PN sequences, or simulate multiframe transmission such as those specified by some mobile communications protocols. As the example in the following figure shows, a GSM multiframe transmission requires 26 frames for speech.

Figure 8-1 GSM Multiframe Transmission



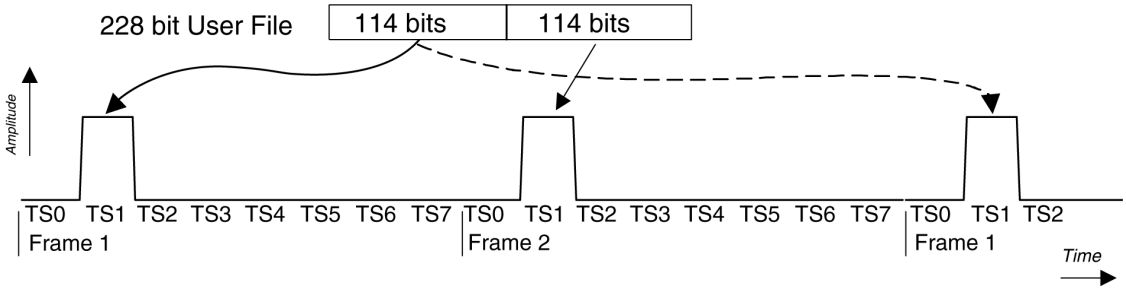
When you select a user file as the data source for a framed transmission, the signal generator's firmware loads PRAM with the framing protocol of the active TDMA format. This creates a file named AUTOGEN_PRAM_1 in addition to a copy of the user file. For all addresses corresponding to active (on) timeslots, the signal generator sets the burst bit to 1 and fills the data fields with the user file data. Other bits are set according to the configuration selected. For inactive (off) timeslots, the signal generator sets the burst control bit to 0, with the data being unspecified.

In the last byte that contains the last user file data bit, the signal generator sets the Pattern Reset bit to 1. This causes the user file data pattern to repeat in the next frame.

NOTE The data in PRAM is static. Firmware writes to PRAM once for the configuration selected and the hardware reads this data repeatedly. Firmware overwrites the volatile PRAM memory to reflect the desired configuration only when the data source or TDMA format changes.

For example, transmitting a 228-bit user file for timeslot #1 (TS1) in a normal GSM transmission creates two frames. Per the standard, a GSM normal channel is 156.25 bits long, with two 57-bit data fields (114 user data bits total per timeslot), and 42 bits for control or signalling purposes.The user file completely fills timeslot #1 for two consecutive frames, and then repeats. The seven remaining timeslots in the GSM frame are off, as shown in Figure 8-2

Figure 8-2 Mapping User File Data to a Single Timeslot



NOTE Compliant with the GSM standard, which specifies 156.25-bit timeslots, the signal generator uses 156-bit timeslots and adds an extra guard bit to every fourth timeslot.

For this protocol configuration, the signal generator's firmware loads PRAM with the bits defined in the following table. (These bits are part of the 32-bit word per frame bit.) The Pattern Reset bit, bit 7, is 0 for frame one and 1 for the last byte of frame two.

Frame	Timeslot	PRAM Word Offset	Data Bits	Burst Bits	Pattern Reset Bit
1	0	0 -155	0/1 (don't care)	0 (off)	0 (off)
1	1 (on)	156 - 311	set by GSM standard (42 bits) & first 114 bits of user file	1 (on)	0
1	2	312 - 467	0/1 (don't care)	0	0
1	3	468 - 624	0/1 (don't care)	0	0
1	4	625 - 780	0/1 (don't care)	0	0
1	5	781 - 936	0/1 (don't care)	0	0
1	6	937 - 1092	0/1 (don't care)	0	0
1	7	1093 - 1249	0/1 (don't care)	0	0
2	0	1250 - 1405	0/1 (don't care)	0	0
2	1 (on)	1406 - 1561	set by GSM standard (42 bits) & remaining bits of user file	1 (on)	0
2	2 through 6	1562 - 2342	0/1 (don't care)	0	0 (off)
2	7	2343 - 2499	0/1 (don't care)	0	1 (1 in offset 2499 only)

Event 1 output is set to 0 or 1 depending on the sync out selection, which enables the EVENT 1 output at either the beginning of the frame, beginning of a specific timeslot, or at all timeslots (SCPI command, `:RADio:GSM:SOUT FRAME|SLOT|ALL`).

Because timeslots are configured and enabled within the signal generator, a user file can be individually assigned to one or more timeslots. A timeslot cannot have more than one data source (PN sequence or user file) specified for it. The amount of user file data that can be mapped into hardware memory depends on both the amount of PRAM available on the baseband generator, and the number and size of each frame. (See "Determining Memory Usage for Custom and TDMA User File Data" on page 16.)

PRAM adds 31 bits to each bit in a frame, which forms 32-bit words.
The following shows how to calculate the amount of PRAM storage space required for a GSM superframe:

Bits per superframe = normal GSM timeslot × timeslot per frame × speech multiframe(TCH) × superframe

size of normal GSM timeslot = 156.25 bits timeslots per frame = 8 timeslots.

speech multiframe(TCH) = 26 frames superframe = 51 speech multiframes

1. Calculate the number of bits in the superframe:

   156.25 × 8 × 26 × 51 = 1,657,500 bits

2. Calculate the size of the PRAM file:

   1,657,500 bits × 4 bytes (32-bit words) = 6,630,000 bytes

3. Calculate how much memory the PRAM file will occupy

   6,630,000 bytes / 1,024 bytes per PRAM block = 6,474.6 memory blocks

4. Round the quotient up to the next integer value

   6,475 blocks × 1,024 bytes per block = 6,630,400 bytes

NOTE For the total PRAM memory usage, be sure to add the number of PRAM blocks that the user file occupies to the PRAM file size. For more information, see "Calculating Volatile Memory (PRAM) Usage for Framed Data" on page 17.

Real-Time Custom High Data Rates

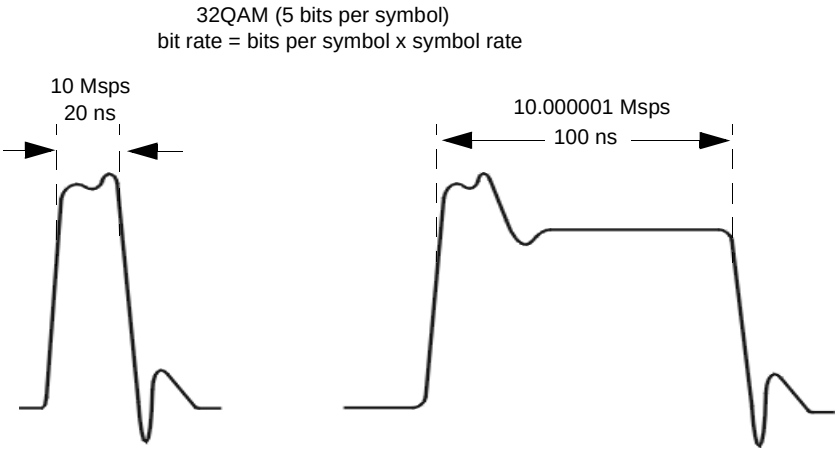
Custom has two modes for processing data, serial and parallel. When the data bit-rate exceeds 50 Mbps, the signal generator processes data in parallel mode, which means processing the data symbol by symbol versus bit by bit (serial). This capability exists in only the Custom format when using a

continuous data stream. This means that it does not apply to a downloaded PRAM file type (covered later in this chapter).

In parallel mode, for a 256QAM modulation scheme, Custom has the capability to reach a data rate of up to 400 Mbps. The FIR filter width is what determines the data rate. The following table shows the maximum data rate for each modulation type. Because the signal generator's maximum symbol rate is 50 Msps, a modulation scheme that has only 1 bit per symbol is always processed in serial mode.

Modulation Type	**Bit Rate Range for Internal Data (bit rate = symbol rate × bits per symbol)**		
	16 Symbol Wide FIR Filter	*32 Symbol Wide FIR Filter*	*64 Symbol Wide FIR Filter*
BPSK, 2FSK, MSK	1bps–50Mbps	1bps–25 Mbps	1bps–12.5Mbps
C4FM, OQPSK, 4FSK	2bps–100Mbps	2bps–50Mbps	2bps–25Mbps
IS95 OQPSK, QPSK			
P4DQPSK, IS95 QPSK			
GRAYQPSK, 4QAM			
D8PSK, EDGE, 8FSK, 8PSK	3bps–150Mbps	3bps–75Mbps	3bps–37.5Mbps
16FSK, 16PSK, 16QAM	4bps–200Mbps	4bps–100Mbps	4bps–50Mbps
Q32AM	5bps–250Mbps	5bps–125Mbps	5bps–62.5Mbps
64QAM	6bps–300Mbps	6bps–150Mbps	6bps–75Mbps
128QAM	7bps–350Mbps	7bps–175Mbps	7bps–87.5Mbps
256QAM	8bps–400Mbps	8bps–200Mbps	8bps–100Mbps

The only external effect of the parallel mode is in the EVENT 1 output signal. In serial and parallel mode, the signal generator outputs a narrow pulse at the EVENT 1 connector. But in parallel mode, the output pulse width increases by a factor of bits-per-symbol wide, as shown in the following figure.



NOTE: The pulse widths values are only for example purposes. The actual width may vary from the above values.

Pattern RAM (PRAM) Data Downloads (E4438C and E8267D)

NOTE This section applies only to the E4438C with Option 001, 002, 601, or 602, and the E8267D with Option 601 or 602.

If you encounter problems with this section, refer to "Data Transfer Troubleshooting (E4438C and E8267D Only)" on page 64.

This section contains information to help you transfer user-generated PRAM data from a system controller to the signal generator's PRAM. It explains how to download data directly into PRAM and modulate the carrier signal with the data.

The control bits included in the PRAM file download, control the following signal functions:

- bursting
- timing signal at the EVENT 1 rear panel connector
- data pattern reset

PRAM data downloads apply to only real-time Custom and TDMA modulation formats. In the TDMA formats, PRAM files are available only while using the unframed data selection. The following table shows which signal generator models support these formats.

E4438C ESG		E2867D PSG
Custom[1]	TDMA[2]	Custom[a]

1. For ESG, requires Option 001, 002, 601, or 602, for PSG requires Option 601 or 602.
2. Real-time TDMA modulation formats require Option 402 and include EDGE, GSM, NADC, PDC, PHS, DECT, and TETRA.

PRAM files differ from bit and binary user files.

Bit and binary user files (see page 10) download to non-volatile memory and the signal generator loads the user file data into PRAM (volatile/waveform memory) for use. The signal generator adds the required control bits when it generates the signal.

A PRAM file downloads directly into PRAM, and it includes seven of the required control bits for each data (payload) bit. The signal generator adds the remaining control bits when it generates the signal. You download the file using either a list or block data format. Programs such as MATLAB or MathCad can generate the data.

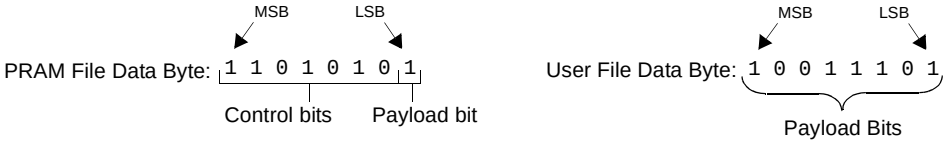
This type of signal control enables you to design experimental or proprietary framing schemes.

After selecting the PRAM file, the signal generator builds the modulation scheme by reading data stored in PRAM, and constructing framing protocols according to the PRAM file data and the modulation format. You can manipulate PRAM data by changing the standard protocols for a modulation format such as the symbol rate, modulation type, and filter either through the front panel interface or with SCPI commands.

Understanding PRAM Files

The term PRAM file comes from earlier Agilent products, the E443xB ESGs. PRAM is another term for waveform memory (WFM1), which is also known as volatile memory. This means that PRAM files and waveform files occupy the same memory location. The signal generator's volatile memory (waveform memory) storage path is `/user/BBG1/waveform`. For more information on memory, see "Signal Generator Memory" on page 3.

The following figure shows a PRAM byte and illustrates the difference between it and a bit/binary user file byte. Notice the control bits in the PRAM byte.



Only three of the seven control bits elicit a response from the signal generator. The other four bits are reserved. Table 9 describes the bits for a PRAM byte.

Table 9 PRAM Data Byte

Bit	Function	Value	Comments
0	Data	0/1	This is the data (payload) bit. It is "unspecified" when burst (bit 2) is set to 0.
1	Reserved	0	Always 0
2	Burst	0/1	1 = RF on 0 = RF off For non-bursted, non-TDMA systems, to have a continuous signal, set this bit to 1 for all bytes. For framed data, set this bit to 1 for *on* timeslots and 0 for *off* timeslots.
3	Reserved	0	Always 0
4	Reserved	1	Always 1
5	Reserved	0	Always 0
6	EVENT1 Output	0/1	To have the signal generator output a single pulse at the EVENT 1 connector, set this bit to 1. Use this output for functions such as a triggering external hardware to indicate when the data pattern begins and restarts, or creating a data-synchronous pulse train by toggling this bit in alternate bytes.
7	Pattern Reset	0/1	0 = continue to next sequential memory address. 1 = end of memory and restart memory playback. This bit is set to 0 for all bytes except the last byte of PRAM. To restart the pattern, set the last byte of PRAM to 1.

As seen in Table 9, only four bits, shown in the following list, can change state:

- bit 0—data
- bit 2—bursting
- bit 6—EVENT 1 rear panel output
- bit 7—pattern reset

Because a PRAM byte has only four bits that can change states, there are only 15 possible byte patterns as shown in Table 10. The table also shows the decimal value for each pattern, which is needed for downloading data using the list format shown on page 37.

Table 10 PRAM Byte Patterns and Bit Positions

Bit Function	Pattern Reset	EVENT 1 Output	Reserved (Bit = 0)	Reserved (Bit = 1)	Reserved (Bit = 0)	Burst	Reserved (Bit = 0)	Data	Bit Pattern Decimal Value
Bit Position	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	---
Bit Pattern	1	1	0	1	0	1	0	1	213
	1	1	0	1	0	1	0	0	212
	1	1	0	1	0	0	0	1	209
	1	1	0	1	0	0	0	0	208
	1	0	0	1	0	1	0	1	149
	1	0	0	1	0	0	0	1	145
	1	0	0	1	0	0	0	0	144
	0	1	0	1	0	1	0	1	85
	0	1	0	1	0	1	0	0	84
	0	1	0	1	0	0	0	1	81
	0	1	0	1	0	0	0	0	80
	0	0	0	1	0	1	0	1	21
	0	0	0	1	0	1	0	0	20
	0	0	0	1	0	0	0	1	17
	0	0	0	1	0	0	0	0	16

Viewing the PRAM Waveform

After the waveform data is written to PRAM, the data pattern can be viewed using an oscilloscope. There is approximately a 12-symbol delay between a state change in the burst bit and the corresponding effect at the RF out. This delay varies with symbol rate and filter settings, and requires compensation to advance the burst bit in the downloaded PRAM file.

PRAM File Size

Because volatile memory resides on the baseband generator (BBG), the maximum PRAM file size depends on the installed baseband generator option, as shown in Table 11.

Table 11 Maximum PRAM User File Size (Payload Bits Only)

Modulation Format	Baseband Generator Option		
	001, 601	002	602
Custom TDMA	8 Mbits[1]	32 Mbits[a]	64 Mbits[a]

1. File size with no other files residing in volatile memory.

The maximum PRAM user file size in the table above refers to the maximum number of payload bits. After downloading, the signal generator translates each downloaded payload bit into a 32-bit word:

- 1 downloaded payload bit
- 7 downloaded control bits as shown in Table 9 on page 34
- 24 bits added by the signal generator

The following table shows the maximum file size after the signal generator has translated the maximum number of payload bits into 32-bit words.

Table 12 Maximum File Size After Downloading

Modulation Format	Baseband Generator Option		
	001, 601	002	602
Custom TDMA	32 MBytes[1]	128 MBytes[a]	256 MBytes[a]

1. File size with no other files residing in volatile memory.

To properly size a PRAM file, you must determine the file size after the 32-bit translation process. The signal generator measures a PRAM file size in units of bytes; each 32-bit word equals 4 bytes.

Determining the File Size

The following example shows how to calculate a downloaded file size using a PRAM file that contains 89 bytes (payload bits plus 7 control bits per payload bit):

89 bytes + [(89 × 24 bits) / 8] = 356 bytes

Because the file downloads one fourth of the translated 32-bit word, another method to calculate the file size is to multiply the downloaded file size by four:

89 bytes × 4 = 356 bytes

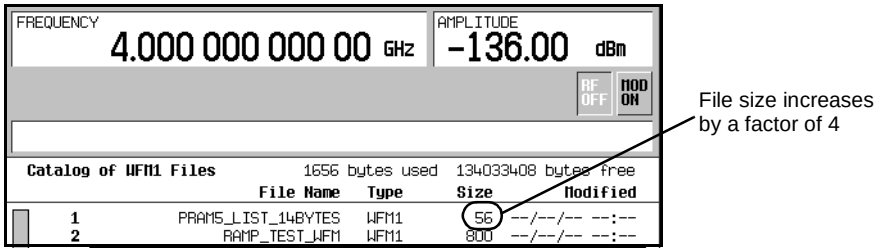
See also “Signal Generator Memory” on page 3 and “Checking Available Memory” on page 8.

Minimum File Size

A PRAM file requires a minimum of 60 bytes to create a signal. If the downloaded file contains less than 60 bytes, the signal generator replicates the file until the file size meets the 60 byte minimum. This replication process occurs after you select the file and turn the modulation format on. The following example shows this process using a downloaded 14-byte file:

- During the file download, the 14 bytes are translated into 56 bytes (fourteen 32-bit words).

  14 bytes × 4 = 56 bytes



- After selecting and turning the format on, the signal generator replicates the file contents to create the 60 byte minimum file size

  60 bytes / 14 bytes = 4.29 file replications

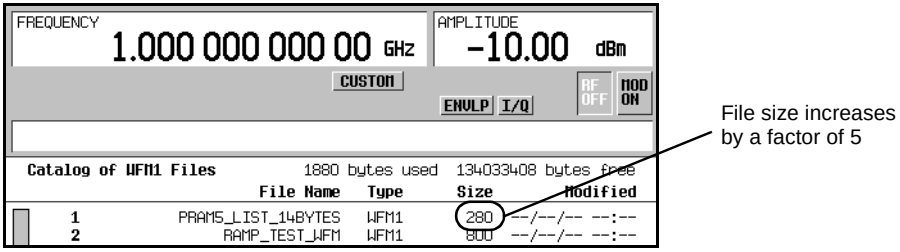
  The signal generator rounds this real value up to the next highest integer. In this example, the signal generator replicates the fourteen 32-bit words (56 bytes) by a factor of 5, which makes the final file size 280 bytes. This equates to a 70 byte file.

  14 bytes × 5 = 70 bytes

  70 + [(70 × 24) / 8] = 280 bytes

  Or

  56 bytes × 5 = 280 bytes



SCPI Command for a List Format Download

Using the list format, enter the data in the command line using comma-separated decimal values. This file type takes longer to download because the signal generator must parse the data. When creating the data, remember that the signal generator requires a minimum of 60 bytes. For more information on file size limits, see "PRAM File Size" on page 36.

Command Syntax

```
:MEMory:DATA:PRAM:FILE:LIST <"file_name">,<uint8>[,<uint8>,<...>]
```

`uint8` The decimal equivalent of an unsigned 8-bit integer value. For a list of usable decimal values and their meaning with respect to the generated signal, see Table 10 on page 35.

Command Syntax Example

The following example, when executed, creates a new file in volatile (waveform) memory with the following attributes:

- creates a file named *new_file*
- outputs a single pulse at the EVENT 1 connector
- bursts the data pattern 1100 seven times over 28 bytes
- transmits 32 nonbursted bytes
- resets the data pattern so it starts again

```
:MEMory:DATA:PRAM:FILE:LIST <"new_file">,85,21,20,20,21,21,20,20,21,21,20,20,21,21,
20,20,21,21,20,20,21,21,20,20,21,21,20,20,16,16,16,16,16,16,16,16,16,16,16,16,16,
16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,144
```

The following list defines the meaning of the different bytes seen in the command line:

85 Send a pulse to the EVENT 1 output, and burst the signal with a data bit of 1.

21 Burst the signal with a data bit of 1.

20 Burst the signal with a data bit of 0.

16 Do not burst the signal (RF output off), and set the data bit to 0.

144 Reset the data pattern, do not burst the signal (RF output off), and set the data bit to 0.

SCPI Command for a Block Data Download

The IEEE standard 488.2-1992 section 7.7.6 defines block data. The signal generator is able to download block data significantly faster than list formatted data (see page 37), because it does not have to parse the data. When creating the data, remember that the signal generator requires a minimum of 60 bytes. For more information on file size limits, see "PRAM File Size" on page 36.

Command Syntax

```
:MEMory:DATA:PRAM:FILE:BLOCk <"file_name">,<blockdata>
```

The following sections explain how to format the SCPI command for downloading block data:

- Command Syntax Example
- Command Syntax in a Program Routine

Command Syntax Example

This example conceptually describes how to format a block data download command (#ABC represents the block data):

```
:MEMory:DATA:PRAM:FILE:BLOCk <"file_name">,#ABC
```

<"file_name">	the file name as it will appear in the signal generator
#	indicates the start of the block data
A	the number of decimal digits present in B
B	a decimal number specifying the number of data bytes to follow in C
C	the PRAM file data in bytes

```
:MEMory:DATA:PRAM:FILE:BLOCk "my_file",#324012%S!4&07#8g*Y9@7...
```

file_name A B C

my_file	the PRAM file name as it will appear in the signal generator's WFM1 memory catalog
#	indicates the start of the block data
3	B has three decimal digits
240	240 bytes of data to follow in C
12%S!4&07#8g*Y9@7...	the ASCII representation of some of the block data (binary data) downloaded to the signal generator, however not all ASCII values are printable

In actual use, the block data is not part of the command line as shown above, but instead resides in a binary file on the PC/UNIX. When the program executes the SCPI command, the command line notifies the signal generator that it is going to receive block data of the stated size, and to place the file in the signal generator file directory with the indicated name. Immediately following the command execution, the program downloads the binary file to the signal generator. This is shown in the following section, "Command Syntax in a Program Routine"

Command Syntax in a Program Routine

This section demonstrates the use of the download SPCI command within the confines of a C++ program routine. The following code sends the SCPI command and downloads a 240 byte PRAM file to the signal generator's WFM1 (waveform) memory catalog. This program assumes that there is a char array, *databuffer*, that contains the 240 bytes of PRAM data and that the variable *numbytes* stores the length of the array.

Line **Code—Download PRAM File Data**

```
1  int bytesToSend;
2  bytesToSend = numbytes;
3  char s[4];
4  char cmd[200];
5  sprintf(s, "%d", bytesToSend);
6  sprintf(cmd, ":MEM:DATA:PRAM:FILE:BLOCk \"FILE1\", #%d%d", strlen(s),bytesToSend);
7  iwrite(id, cmd, strlen(cmd), 0, 0);
8  iwrite(id, databuffer, bytesToSend, 0, 0);
9  iwrite(id, "\n", 1, 1, 0);
```

Line	Code Description—Download PRAM File Data
1	Define an integer variable (*bytesToSend*) to store the number of bytes to send to the signal generator.
2	Store the total number of PRAM bytes in the integer variable defined in line 1. *numbytes* contains the length of the *databuffer* array referenced in line 8.
3	Create a string large enough to hold the *bytesToSend* value as characters plus a null character value. In this code, string *s* is set to 4 bytes (3 characters for the bytesToSend value and one null character—one character equals one byte).
4	Create a string and set its length (*cmd*[200]) to hold the SCPI command syntax and parameters. In this code, we define the string length as 200 bytes (200 characters).
5	Store the value of *bytesToSend* in string *s*. For this example, bytesToSend = 240; *s* = "240"
6	Store the SCPI command syntax and parameters in the string *cmd*. The SCPI command prepares the signal generator to accept the data. • *sprintf()* is a standard function in C++, which writes string data to a string variable. • *strlen()* is a standard function in C++, which returns length of a string. • *bytesToSend* = 240, then *s* = "240" plus the null character, *strlen(s)* = 4, so *cmd* = :MEM:DATA:PRAM:FILE:BLOCk "FILE1\" #3240.
7	Send the SCPI command stored in the string *cmd* to the signal generator contained in the variable *id*. • *iwrite()* is a SICL function in Agilent IO library, which writes the data (block data) specified in the string *cmd* to the signal generator. • The third argument of *iwrite()*, *strlen(cmd)*, informs the signal generator of the number of bytes in the command string. The signal generator parses the string to determine the number of data bytes it expects to receive. • The fourth argument of iwrite(), 0, means there is no END of file indicator for the string. This lets the session remain open, so the program can download the PRAM file data.

Line	Code Description—Download PRAM File Data
8	Send the PRAM file data stored in the array, *databuffer*, to the signal generator. • *iwrite()* sends the data specified in *databuffer* (PRAM data) to the signal generator (session identifier specified in *id*). • The third argument of *iwrite()*, *bytesToSend*, contains the length of the *databuffer* in bytes. In this example, it is 240. • The fourth argument of *iwrite()*, 0, means there is no END of file indicator in the data. In many programming languages, there are two methods to send SCPI commands and data: — Method 1 where the program stops the data download when it encounters the first zero (END indicator) in the data. — Method 2 where the program sends a fixed number of bytes and ignores any zeros in the data. This is the method used in our program. For your programming language, you must find and use the equivalent of method two. Otherwise you may only achieve a partial download of the user file data.
9	Send the terminating carriage (\n) as the last byte of the waveform data. • *iwrite()* writes the data "\n" to the signal generator (session identifier specified in *id).* • The third argument of *iwrite()*, 1, sends one byte to the signal generator. • The fourth argument of *iwrite(),* 1, is the END of file indicator, which the program uses to terminate the data download.

Selecting a Downloaded PRAM File as the Data Source

The following steps show the process for selecting a PRAM file using commands from the GSM (TDMA) modulation format. While the commands shown come from only one format, the concept remains the same when making the data selection for any of the other real-time modulation formats that support PRAM data. To find the commands for Custom and the other TDMA formats, see the signal generator's *SCPI Command Reference*.

1. For real-time TDMA formats, select unframed data:

   ```
   :RADio:GSM:BURSt:STATe OFF
   ```

2. Select the data type:

   ```
   :RADio:GSM:DATA PRAM
   ```

3. Select the PRAM file:

   ```
   :RADio:GSM:DATA:PRAM <"file_name">
   ```

   Because the command is file specific (PRAM), there is no need to include the file path with the file name.

4. Configure the remaining signal parameters.
5. Turn the modulation format on:

```
:RADio:GSM:STATe On
```

Modulating and Activating the Carrier

Use the following commands to modulate the carrier and turn on the RF output. For a complete listing of SPCI commands, refer to the *SCPI Command Reference*.

```
:FREQuency:FIXed 1.8GHZ
:POWer:LEVel -10.0DBM
:OUTPut:MODulation:STATe ON
:OUTPut:STATe ON
```

Storing a PRAM File to Non-Volatile Memory and Restoring to Volatile Memory

After you download the file to volatile memory (waveform memory), you can then save it to non-volatile memory. Remember that a PRAM file downloads to waveform memory. Conversely, when you store a PRAM file to non-volatile memory, it uses the same directory as waveform files. When storing or restoring a file, you must include the file path as part of the file_name variable.

Command Syntax

The first file_name variable is the current location of the file and its name; the second file_name variable is the destination to store the file and its name.

There are three ways to format the file_name variable to include the file path:

Volatile Memory to Non-Volatile Memory

```
:MEMory:COPY "WFM1:file_name","NVWFM:file_name"
:MEMory:COPY "file_name@WFM1","file_name@NVWFM"
:MEMory:COPY "/user/bbg1/waveform/file_name","/user/waveform/file_name"
```

Non-Volatile Memory to Volatile Memory

```
:MEMory:COPY "NVWFM:file_name","WFM1:file_name"
:MEMory:COPY "file_name@NVWFM","file_name@WFM1"
:MEMory:COPY "/user/waveform/file_name","/user/bbg1/waveform/file_name"
```

Extracting a PRAM File

When you extract a PRAM file, you are extracting the translated 32-bit word-per-byte file. You cannot extract just the downloaded data. Extracting a PRAM file is similar to extracting a waveform file in that you use the same commands, and the PRAM file resides in either volatile memory (waveform memory) or the waveform directory for non-volatile memory. After extraction, you can download the file to the same signal generator or to another signal generator with the proper option configuration that supports the downloaded file. There are two ways to download a file after extraction:

- with the ability to extract later
- with no extraction capability

CAUTION Ensure that you do not use the `:MEMory:DATA:PRAM:FILE:BLOCk` command to download an extracted file. If you use this command, the signal generator will treat the file as a new PRAM file and translate the LSB of each byte into a 32-bit word, corrupting the file data.

Command Syntax

This section lists the commands for extracting PRAM files and downloading extracted PRAM files. To download an extracted file, you must use block data. For information on block data, see "SCPI Command for a Block Data Download" on page 38. In addition, there are three ways to format the file_name variable, which must also include the file path, as shown in the following tables.

There are two commands for file extraction:

- `:MEM:DATA? <"file_name">`
- `:MMEM:DATA? <"filename">`

The following table uses the first command to illustrate the command format, however the format is the same if you use the second command.

Table 13 Extracting a PRAM File

Extraction Method/Memory Type	Command Syntax Options

SCPI/volatile memory	`:MEM:DATA? "WFM1:file_name"`
	`:MEM:DATA? "file_name@WFM1"`
	`:MEM:DATA? "/user/bbg1/waveform/file_name"`

SCPI/non-volatile memory	`:MEM:DATA? "NVWFM:file_name"`
	`:MEM:DATA? "file_name@NVWFM"`
	`:MEM:DATA? "/user/waveform/file_name"`

FTP/volatile memory[1]	`get /user/bbg1/waveform/file_name`
FTP/non-volatile memory[a]	`get /user/waveform/file_name`

1. See "FTP Procedures" on page 26.

Table 14 Downloading a File for Extraction

Download Method/ Memory Type	Command Syntax Options

SCPI/volatile memory	`:MEM:DATA:UNPRotected "WFM1:file_name",<blockdata>`
	`:MEM:DATA:UNPRotected "file_name@WFM1",<blockdata>`
	`:MEM:DATA:UNPRotected "/user/bbg1/waveform/file_name",<blockdata>`

Table 14 Downloading a File for Extraction

Download Method/ Memory Type	Command Syntax Options
SCPI/non-volatile memory	:MEM:DATA:UNPRotected "NVWFM:file_name",<blockdata> :MEM:DATA:UNPRotected "file_name@NVWFM",<blockdata> :MEM:DATA:UNPRotected "/user/waveform/file_name",<blockdata>
FTP/volatile memory[1]	put <file_name> /user/bbg1/waveform/file_name
FTP/non-volatile memory[a]	put <file_name> /user/waveform/file_name

1. See "FTP Procedures" on page 26.

There are two commands that download a file for no extraction:

- :MEM:DATA <"file_name">,<blockdata>
- :MMEM:DATA <"filename">,<blockdata>

The following table uses the first command to illustrate the command format, however the format is the same if you use the second command.

Table 15 Downloading a File for No Extraction

Download Method/ Memory Type	Command Syntax Options
SCPI/volatile memory	:MEM:DATA "WFM1:file_name",<blockdata> :MEM:DATA "file_name@WFM1",<blockdata> :MMEM:DATA "user/bbg1/waveform/file_name",<blockdata>
SCPI/non-volatile memory	:MEM:DATA "NVWFM:file_name",<blockdata> :MEM:DATA "file_name@NVWFM",<blockdata> :MEM:DATA /user/waveform/file_name",<blockdata>

Modifying PRAM Files

The only way to change PRAM file data is to modify the original file on a computer and download it again. The signal generator does not support viewing and editing PRAM file contents. Because the signal generator translates the data bit into a 32-bit word, the file contents are not recognizable, and therefore not editable using a hex editor program, as shown in the following figure.

60 byte PRAM file prior to downloading

```
00000000:  85 15 15 15 15 14 15 14 15 14 14 15 15 15 14 14
00000010:  14 15 15 15 14 15 14 14 15 14 14 15 15 14 14 15
00000020:  15 14 14 14 14 14 15 15 14 14 15 15 14 15 15 14
00000030:  14 14 15 14 14 15 15 15 15 15 14 90 __
```

60 byte PRAM file after downloading

```
00000000:  00 01 01 40 00 01 00 40 00 01 00 40 00 01 00 40
00000010:  00 01 00 40 00 00 00 40 00 01 00 40 00 00 00 40
00000020:  00 01 00 40 00 00 00 40 00 00 00 40 00 01 00 40
00000030:  00 01 00 40 00 01 00 40 00 00 00 40 00 00 00 40
00000040:  00 00 00 40 00 01 00 40 00 01 00 40 00 01 00 40
00000050:  00 00 00 40 00 01 00 40 00 00 00 40 00 00 00 40
00000060:  00 01 00 40 00 00 00 40 00 00 00 40 00 01 00 40
00000070:  00 01 00 40 00 00 00 40 00 00 00 40 00 01 00 40
00000080:  00 01 00 40 00 00 00 40 00 00 00 40 00 00 00 40
00000090:  00 00 00 40 00 00 00 40 00 01 00 40 00 01 00 40
000000a0:  00 00 00 40 00 00 00 40 00 01 00 40 00 01 00 40
000000b0:  00 00 00 40 00 01 00 40 00 01 00 40 00 00 00 40
000000c0:  00 00 00 40 00 00 00 40 00 01 00 40 00 00 00 40
000000d0:  00 00 00 40 00 01 00 40 00 01 00 40 00 01 00 40
000000e0:  00 01 00 40 00 01 00 40 00 00 00 40 00 00 00 00
000000f0:  __
```

FIR Filter Coefficient Downloads (E4438C and E8267D)

NOTE If you encounter problems with this section, refer to "Data Transfer Troubleshooting (E4438C and E8267D Only)" on page 64.

The signal generator accepts finite impulse response (FIR) filter coefficient downloads. After downloading the coefficients, these user-defined FIR filter coefficient values can be selected as the filtering mechanism for the active digital communications standard.

Data Requirements

There are two requirements for user-defined FIR filter coefficient files:

1. Data must be in ASCII format.

   The signal generator processes FIR filter coefficients as floating point numbers.

2. Data must be in List format.

   FIR filter coefficient data is processed as a list by the signal generator's firmware. See Sample Command Line.

Data Limitations

Filter lengths of up to 1024 taps (coefficients) are allowed. The oversample ratio (OSR) is the number of filter taps per symbol. Oversample ratios from 1 through 32 are possible.

The maximum combination of OSR and symbols allowed is 32 symbols with an OSR of 32.

The Real Time I/Q Baseband FIR filter files are limited to 1024 taps, 64 symbols and a 16-times oversample ratio. FIR filter files with more than 64 symbols cannot be used.

The ARB Waveform Generator FIR filter files are limited to 512 taps and 512 symbols.

The sampling period (Δt) is equal to the inverse of the sampling rate (FS). The sampling rate is equal to the symbol rate multiplied by the oversample ratio. For example, the GSM symbol rate is 270.83 ksps. With an oversample ratio of 4, the sampling rate is 1083.32 kHz and Δt (inverse of FS) is 923.088 nsec.

Downloading FIR Filter Coefficient Data

The signal generator stores the FIR files in the FIR (/USER/FIR) directory, which utilizes non-volatile memory (see also "Signal Generator Memory" on page 3). Use the following SCPI command line to download FIR filter coefficients (file) from the PC to the signal generator's FIR directory:

```
:MEMory:DATA:FIR <"file_name">,osr,coefficient{,coefficient}
```

Use the following SCPI command line to query list data from the FIR file:

```
:MEMory:DATA:FIR? <"file_name">
```

Sample Command Line

The following SCPI command will download a typical set of FIR filter coefficient values and name the file “FIR1”:

```
:MEMory:DATA:FIR "FIR1",4,0,0,0,0,0,0.000001,0.000012,0.000132,0.001101,
0.006743,0.030588,0.103676,0.265790,0.523849,0.809508,1,1,0.809508,0.523849,
0.265790,0.103676,0.030588,0.006743,0.001101,0.000132,0.000012,0.000001,0,
0,0,0,0
```

`FIR1`	assigns the name FIR1 to the associated OSR (over sample ratio) and coefficient values (the file is then represented with this name in the FIR File catalog)
4	specifies the oversample ratio
`0,0,0,0,0,` `0.000001,...`	the FIR filter coefficients

Selecting a Downloaded User FIR Filter as the Active Filter

NOTE For information on manual key presses for the following remote procedures, refer to the *User’s Guide*.

FIR Filter Data for TDMA Format

The following remote command selects user FIR filter data as the active filter for a TDMA modulation format.

```
:RADio:<desired format>:FILTer <"file_name">
```

This command selects the user FIR filter, specified by the file name, as the active filter for the TDMA modulation format. After selecting the file, activate the TDMA format with the following command:

```
:RADio:<desired format>:STATe On
```

FIR Filter Data for Custom Modulation

The following remote command selects user FIR filter data as the active filter for a custom modulation format.

```
:RADio:CUSTom:FILTer <"file_name">
```

This command selects the user FIR filter, specified by the file name, as the active filter for the custom modulation format. After selecting the file, activate the TDMA format with the following command:

```
:RADio:CUSTom:STATe On
```

FIR Filter Data for CDMA and W-CDMA Modulation

The following remote command selects user FIR filter data as the active filter for a CDMA modulation format. The process is very similar for W-CDMA.

```
:RADio:<desired format>:ARB:FILTer <"file_name">
```

This command selects the User FIR filter, specified by the file name, as the active filter for the CDMA or W-CDMA modulation format. After selecting the file, activate the CDMA or W-CDMA format with the following command:

```
:RADio:<desired format>:ARB:STATe On
```

Modulating and Activating the Carrier

The following commands set the carrier frequency and power, and turns on the modulation and the RF output.

1. Set the carrier frequency to 2.5 GHz:

   ```
   :FREQuency:FIXed 2.5GHZ
   ```

2. Set the carrier power to –10.0 dBm:

   ```
   :POWer:LEVel -10.0DBM
   ```

3. Activate the modulation:

   ```
   :OUTPut:MODulation:STATe ON
   ```

4. Activate the RF output:

   ```
   :OUTPut:STATe ON
   ```

Save and Recall Instrument State Files

NOTE References to waveform files and some of the other data file types mentioned in the following sections are not available for all models and options of signal generator. Refer to the instrument's *Data Sheet* for the signal generator and options being used.

The signal generator can save instrument state settings to memory. An instrument state setting includes any instrument state that does not survive a signal generator preset or power cycle such as frequency, amplitude, attenuation, and other user–defined parameters. The instrument state settings are saved in memory and organized into sequences and registers. There are 10 sequences with 100 registers per sequence available for instrument state settings. These instrument state files are stored in the USER/STATE directory. See also, "Signal Generator Memory" on page 3.

The save function does not store data such as Arb waveforms, table entries, list sweep data, and so forth. The save function saves a reference to the waveform or data file name associated with the instrument state. Use the store commands or store softkey functions to store these data file types to the signal generator's memory catalog.

Before saving an instrument state that has a data file or waveform file associated with it, store the file. For example, if you are editing a multitone arb format, store the multitone data to a file in the signal generator's memory catalog (multitone files are stored in the USER/MTONE directory). Then save the instrument state associated with that data file. The settings for the signal generator such as frequency and amplitude and a reference to the multitone file name will be saved in the selected sequence and register number. Refer to the signal generator's *User's Guide*, *Key and Data Field Reference,* or the signal generator's Help hardkey for more information on the save and recall functions.

Save and Recall SCPI Commands

The following command sequence saves the current instrument state, using the *SAV command, in register 01, sequence 1. A comment is then added to the instrument state.

```
*SAV 01,1
:MEM:STAT:COMM 01,1,"Instrument state comment"
```

If there is a waveform or data file associated with the instrument state, there will be a file name reference saved along with the instrument state. However, the waveform/data file must be stored in the signal generator's memory catalog as the *SAV command does not save data files. For more information on storing file data such as modulation formats, arb setups, and table entries refer to the signal generator's *User's Guide*.

NOTE On the N5182A, E4438C, and E8267D, if a saved instrument state contains a reference to a waveform file, ensure that the waveform file resides in volatile memory before recalling the instrument state. For more information, see the *User's Guide*.

The recall function recalls a saved instrument state. If there is a data file associated with the instrument state, the file will be loaded along with the instrument state. The following command

recalls the instrument state saved in register 01, sequence 1.

```
*RCL 01,1
```

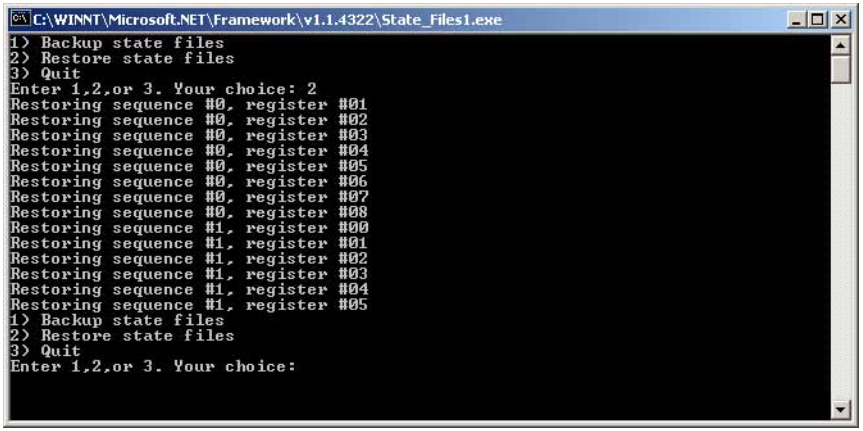
Save and Recall Programming Example Using VISA and C#

The following programming example uses VISA and C# to save and recall signal generator instrument states. Instruments states are saved to and recalled from your computer. This console program prompts the user for an action: Backup State Files, Restore State Files, or Quit.

The Backup State Files choice reads the signal generator's state files and stores it on your computer in the same directory where the State_Files.exe program is located. The Restore State Files selection downloads instrument state files, stored on your computer, to the signal generator's State directory. The Quit selection exists the program. The figure below shows the console interface and the results obtained after selecting the Restore State Files operation.

The program uses VISA library functions. Refer to the Agilent VISA User's Manual available on Agilent's website: *http:\\www.agilent.com* for more information on VISA functions.

The program listing for the State_Files.cs program is shown below. It is available on the CD–ROM in the programming examples section under the same name.



C# and Microsoft .NET Framework

The Microsoft .NET Framework is a platform for creating Web Services and applications. There are three components of the .NET Framework: the common language runtime, class libraries, and Active Server Pages, called ASP.NET. Refer to the Microsoft website for more information on the .NET Framework.

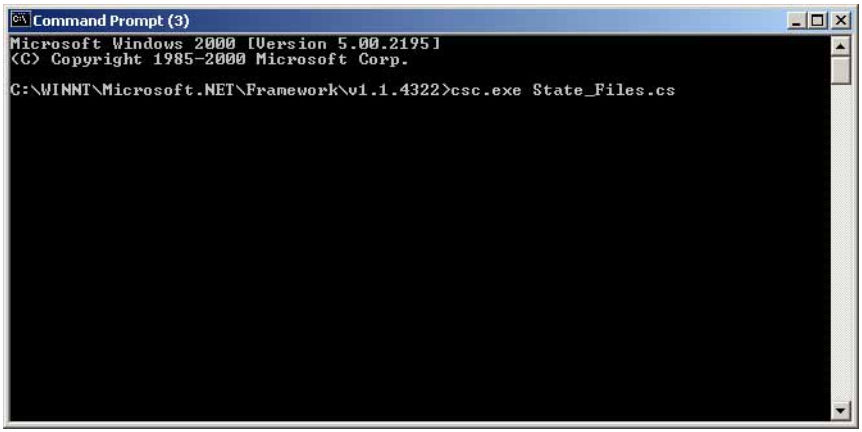
The .NET Framework must be installed on your computer before you can run the State_Files program. The framework can be downloaded from the Microsoft website and then installed on your computer.

Perform the following steps to run the State_Files program.

1. Copy the State_Files.cs file from the CD–ROM programming examples section to the directory

where the .NET Framework is installed.

2. Change the TCPIP0 address in the program from TCPIP0::000.000.000.000 to your signal generator's address.
3. Save the file using the .cs file name extension.
4. Run the Command Prompt program. Start > Run > "cmd.exe". Change the directory for the command prompt to the location where the .NET Framework was installed.
5. Type csc.exe State_Files.cs at the command prompt and then press the Enter key on the keyboard to run the program. The following figure shows the command prompt interface.



The State_Files.cs program is listed below. You can copy this program from the examples directory on the signal generator's CD–ROM.

NOTE The *State_Files.cs* example uses the ESG in the programming code but can be used with the PSG or Agilent MXG.

```
//**************************************************************************************
// FileName: State_Files.cs
//
// This C# example code saves and recalls signal generator instrument states. The saved
// instrument state files are written to the local computer directory computer where the
// State_Files.exe is located. This is a console application that uses DLL importing to
// allow for calls to the unmanaged Agilent IO Library VISA DLL.
//
// The Agilent VISA library must be installed on your computer for this example to run.
// Important: Replace the visaOpenString with the IP address for your signal generator.
```

```
//
//*************************************************************************************
using System;
using System.IO;
using System.Text;
using System.Runtime.InteropServices;
using System.Collections;
using System.Text.RegularExpressions;

namespace State_Files

{
    class MainApp
    {
        // Replace the visaOpenString variable with your instrument's address.

        static public string visaOpenString = "TCPIP0::000.000.000.000"; //"GPIB0::19";
      //"TCPIP0::ESG3::INSTR";


public const uint DEFAULT_TIMEOUT = 30 * 1000;// Instrument timeout 30 seconds.
        public const int MAX_READ_DEVICE_STRING = 1024; // Buffer for string data reads.
      public const int TRANSFER_BLOCK_SIZE = 4096;// Buffer for byte data.

        // The main entry point for the application.

        [STAThread]

static void Main(string[] args)
        {

    uint defaultRM;// Open the default VISA resource manager
    if (VisaInterop.OpenDefaultRM(out defaultRM) == 0) // If no errors, proceed.
      {
         uint device;
         // Open the specified VISA device: the signal generator
     if (VisaInterop.Open(defaultRM, visaOpenString,VisaAccessMode.NoLock,
                          DEFAULT_TIMEOUT, out device) == 0)
        // if no errors proceed.
       {
       bool quit = false;
         while (!quit)// Get user input
          {
```

```
  Console.Write("1) Backup state files\n" +
                "2) Restore state files\n" +
                "3) Quit\nEnter 1,2,or 3. Your choice: ");
     string choice = Console.ReadLine();
   switch (choice)
        {
         case "1":
         {
       BackupInstrumentState(device);  // Write instrument state
       break;                   // files to the computer
         }
                    case "2":
         {
        RestoreInstrumentState(device); // Read instrument state
        break;// files to the sig gen
         }
      case "3":
         {
       quit = true;
       break;
         }
       default:
         {
       break;
         }
        }
       }
    VisaInterop.Close(device);// Close the device
       }
     else
       {
     Console.WriteLine("Unable to open " + visaOpenString);
    }
        VisaInterop.Close(defaultRM);   // Close the default resource manager
    }
     else
       {
  Console.WriteLine("Unable to open the VISA resource manager");
    }
  }

/* This method restores all the sequence/register state files located in
 the local directory (identified by a ".STA" file name extension)
 to the signal generator.*/
```

```
static public void RestoreInstrumentState(uint device)
   {
    DirectoryInfo di = new DirectoryInfo(".");// Instantiate object class
    FileInfo[] rgFiles = di.GetFiles("*.STA");  // Get the state files
    foreach(FileInfo fi in rgFiles)
      {
      Match m = Regex.Match(fi.Name, @"^(\d)_(\d\d)");
      if (m.Success)
        {
        string sequence = m.Groups[1].ToString();
        string register = m.Groups[2].ToString();
        Console.WriteLine("Restoring sequence #" + sequence +
                                    ", register #" + register);

/* Save the target instrument's current state to the specified sequence/
register pair. This ensures the index file has an entry for the specified
sequence/register pair. This workaround will not be necessary in future
revisions of firmware.*/

        WriteDevice(device,"*SAV " + register + ", " + sequence + "\n",
                              true); // << on SAME line!
        // Overwrite the newly created state file with the state
        // file that is being restored.
        WriteDevice(device, "MEM:DATA \"/USER/STATE/" + m.ToString() + "\",",
                              false); // << on SAME line!
        WriteFileBlock(device, fi.Name);
        WriteDevice(device, "\n", true);
                 }
               }
        }

/* This method reads out all the sequence/register state files from the signal
generator and stores them in your computer's local directory with a ".STA"
extension */

static public void BackupInstrumentState(uint device)
   {
   // Get the memory catalog for the state directory
      WriteDevice(device, "MEM:CAT:STAT?\n", false);
      string catalog = ReadDevice(device);
      /* Match the catalog listing for state files which are named
         (sequence#)_(register#)  e.g.  0_01, 1_01, 2_05*/
```

```
      Match m = Regex.Match(catalog, "\"(\\d_\\d\\d),");
      while (m.Success)
         {
        // Grab the matched filename from the regular expresssion
        string nextFile = m.Groups[1].ToString();
       // Retrieve the file and store with a .STA extension
       // in the current directory
       Console.WriteLine("Retrieving state file: " + nextFile);
         WriteDevice(device, "MEM:DATA? \"/USER/STATE/" + nextFile + "\"\n", true);
       ReadFileBlock(device, nextFile + ".STA");
       // Clear newline
       ReadDevice(device);
       // Advance to next match in catalog string
       m = m.NextMatch();
        }
   }

/*  This method writes an ASCII text string (SCPI command) to the signal generator.
If the bool "sendEnd" is true, the END line character will be sent at the
conclusion of the write. If "sendEnd is false the END line will not be sent.*/

static public void WriteDevice(uint device, string scpiCmd, bool sendEnd)
   {
    byte[] buf = Encoding.ASCII.GetBytes(scpiCmd);
    if (!sendEnd) // Do not send the END line character
      {
     VisaInterop.SetAttribute(device, VisaAttribute.SendEndEnable, 0);
      }
    uint retCount;
    VisaInterop.Write(device, buf, (uint)buf.Length, out retCount);
    if (!sendEnd) // Set the bool sendEnd true.
      {
     VisaInterop.SetAttribute(device, VisaAttribute.SendEndEnable, 1);
      }
   }

// This method reads an ASCII string from the specified device
static public string ReadDevice(uint device)
   {
    string retValue = "";
    byte[] buf = new byte[MAX_READ_DEVICE_STRING]; // 1024 bytes maximum read
    uint retCount;
```

```
    if (VisaInterop.Read(device, buf, (uint)buf.Length -1, out retCount) == 0)
      {
      retValue = Encoding.ASCII.GetString(buf, 0, (int)retCount);
      }
    return retValue;
   }

/* The following method reads a SCPI definite block from the signal generator
and writes the contents to a file on your computer. The trailing
newline character is NOT consumed by the read.*/

static public void ReadFileBlock(uint device, string fileName)
   {
    // Create the new, empty data file.
    FileStream fs = new FileStream(fileName, FileMode.Create);
    // Read the definite block header:  #{lengthDataLength}{dataLength}
    uint retCount = 0;
    byte[] buf = new byte[10];
    VisaInterop.Read(device, buf, 2, out retCount);
    VisaInterop.Read(device, buf, (uint)(buf[1]-'0'), out retCount);
     uint fileSize = UInt32.Parse(Encoding.ASCII.GetString(buf, 0, (int)retCount));
    // Read the file block from the signal generator
    byte[] readBuf = new byte[TRANSFER_BLOCK_SIZE];
    uint bytesRemaining = fileSize;

    while (bytesRemaining != 0)
      {
       uint bytesToRead = (bytesRemaining < TRANSFER_BLOCK_SIZE) ?
       bytesRemaining : TRANSFER_BLOCK_SIZE;
       VisaInterop.Read(device, readBuf, bytesToRead, out retCount);
       fs.Write(readBuf, 0, (int)retCount);
       bytesRemaining -= retCount;
      }
    // Done with file
    fs.Close();
   }

/* The following method writes the contents of the specified file to the
specified file in the form of a SCPI definite block.  A newline is
NOT appended to the block and END is not sent at the conclusion of the
write.*/
```

```
static public void WriteFileBlock(uint device, string fileName)
   {
    // Make sure that the file exists, otherwise sends a null block
    if (File.Exists(fileName))
     {
     FileStream fs = new FileStream(fileName, FileMode.Open);
     // Send the definite block header: #{lengthDataLength}{dataLength}
     string fileSize = fs.Length.ToString();
     string fileSizeLength = fileSize.Length.ToString();
     WriteDevice(device, "#" + fileSizeLength + fileSize, false);
     // Don't set END at the end of writes
     VisaInterop.SetAttribute(device, VisaAttribute.SendEndEnable, 0);
     // Write the file block to the signal generator
     byte[] readBuf = new byte[TRANSFER_BLOCK_SIZE];
     int numRead = 0;
     uint retCount = 0;
     while ((numRead = fs.Read(readBuf, 0, TRANSFER_BLOCK_SIZE)) != 0)
        {
        VisaInterop.Write(device, readBuf, (uint)numRead, out retCount);
        }
       // Go ahead and set END on writes
     VisaInterop.SetAttribute(device, VisaAttribute.SendEndEnable, 1);
     // Done with file
     fs.Close();
     }
     else
     {
     // Send an empty definite block
     WriteDevice(device, "#10", false);
     }
    }
   }

// Declaration of VISA device access constants
public enum VisaAccessMode
    {
        NoLock = 0,
        ExclusiveLock = 1,
        SharedLock = 2,
        LoadConfig = 4
    }
```

```
// Declaration of VISA attribute constants
public enum VisaAttribute
    {
        SendEndEnable = 0x3FFF0016,
        TimeoutValue  = 0x3FFF001A
    }

// This class provides a way to call the unmanaged Agilent IO Library VISA C
// functions from the C# application

public class VisaInterop
    {
        [DllImport("agvisa32.dll", EntryPoint="viClear")]
        public static extern int Clear(uint session);

        [DllImport("agvisa32.dll", EntryPoint="viClose")]
        public static extern int Close(uint session);

        [DllImport("agvisa32.dll", EntryPoint="viFindNext")]
        public static extern int FindNext(uint findList, byte[] desc);

        [DllImport("agvisa32.dll", EntryPoint="viFindRsrc")]
        public static extern int FindRsrc(
            uint session,
            string expr,
            out uint findList,
            out uint retCnt,
            byte[] desc);

        [DllImport("agvisa32.dll", EntryPoint="viGetAttribute")]
public static extern int GetAttribute(uint vi, VisaAttribute attribute, out uint attrState);

        [DllImport("agvisa32.dll", EntryPoint="viOpen")]
        public static extern int Open(
            uint session,
            string rsrcName,
            VisaAccessMode accessMode,
            uint timeout,
            out uint vi);

        [DllImport("agvisa32.dll", EntryPoint="viOpenDefaultRM")]
        public static extern int OpenDefaultRM(out uint session);
```

```
        [DllImport("agvisa32.dll", EntryPoint="viRead")]
        public static extern int Read(
            uint session,
            byte[] buf,
            uint count,
            out uint retCount);

        [DllImport("agvisa32.dll", EntryPoint="viSetAttribute")]
public static extern int SetAttribute(uint vi, VisaAttribute attribute, uint attrState);

        [DllImport("agvisa32.dll", EntryPoint="viStatusDesc")]
        public static extern int StatusDesc(uint vi, int status, byte[] desc);

        [DllImport("agvisa32.dll", EntryPoint="viWrite")]
        public static extern int Write(
            uint session,
            byte[] buf,
            uint count,
            out uint retCount);
    }
}
```

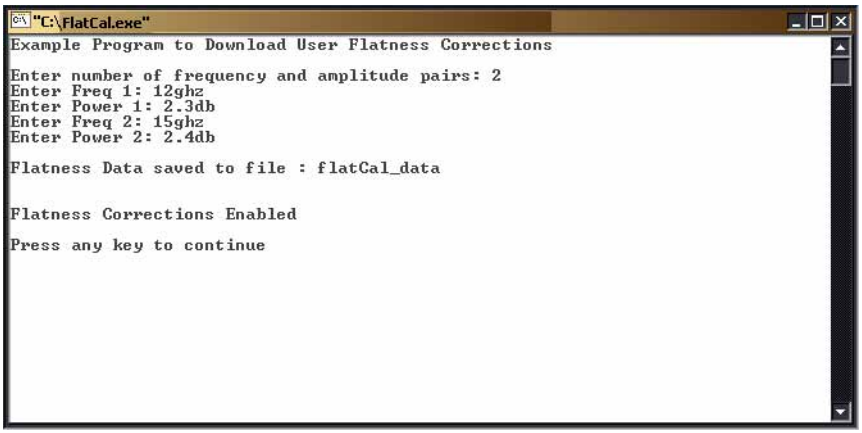
User Flatness Correction Downloads Using C++ and VISA

This sample program uses C++ and the VISA libraries to download user–flatness correction values to the signal generator. The program uses the LAN interface but can be adapted to use the GPIB interface by changing the address string in the program.

You must include header files and resource files for library functions needed to run this program. Refer to the *Programming Guide* for more information.

The FlatCal program asks the user to enter a number of frequency and amplitude pairs. Frequency and amplitude values are entered via the keyboard and displayed on the console interface. The values are then downloaded to the signal generator and stored to a file named flatCal_data. The file is then loaded into the signal generator's memory catalog and corrections are turned on. The figure below shows the console interface and several frequency and amplitude values. Use the same format, shown in the figure below, for entering frequency and amplitude pairs (for example, `12ghz`, `1.2db`).

Figure 15-1 FlatCal Console Application



The program uses VISA library functions. The non–formatted viWrite VISA function is used to output data to the signal generator. Refer to the Agilent VISA User's Manual available on Agilent's website: *http:\\www.agilent.com* for more information on VISA functions.

The program listing for the FlatCal program is shown below. It is available on the CD–ROM in the programming examples section as flatcal.cpp.

```
//*****************************************************************************
// PROGRAM NAME:FlatCal.cpp
//
// PROGRAM DESCRIPTION:C++ Console application to input frequency and amplitude
// pairs and then download them to the signal generator.
//
// NOTE: You must have the Agilent IO Libraries installed to run this program.
//
// This example uses the LAN/TCPIP interface to download frequency and amplitude
// correction pairs to the signal generator. The program asks the operator to enter
// the number of pairs and allocates a pointer array listPairs[] sized to the number
// of pairs.The array is filled with frequency nextFreq[] and amplitude nextPower[]
// values entered from the keyboard.
//
//*****************************************************************************
// IMPORTANT: Replace the 000.000.000.000 IP address in the instOpenString declaration
// in the code below with the IP address of your signal generator.
//*****************************************************************************

#include <stdlib.h>
#include <stdio.h>
#include  "visa.h"
#include <string.h>

//   IMPORTANT:
//   Configure the following IP address correctly before compiling and running

char* instOpenString ="TCPIP0::000.000.000.000::INSTR";//your PSG's IP address

const int MAX_STRING_LENGTH=20;//length of frequency and power strings
const int BUFFER_SIZE=256;//length of SCPI command string

int main(int argc, char* argv[])
{
    ViSession defaultRM, vi;
    ViStatus status = 0;

    status = viOpenDefaultRM(&defaultRM);//open the default resource manager

    //TO DO: Error handling here

    status = viOpen(defaultRM, instOpenString, VI_NULL, VI_NULL, &vi);
```

```
 if (status)//if any errors then display the error and exit the program
 {
     fprintf(stderr, "viOpen failed (%s)\n", instOpenString);
     return -1;
 }

printf("Example Program to Download User Flatness Corrections\n\n");
printf("Enter number of frequency and amplitude pairs: ");
 int num = 0;

 scanf("%d", &num);

 if (num > 0)
 {
     int lenArray=num*2;//length of the pairsList[] array. This array
     //will hold the frequency and amplitude arrays

     char** pairsList = new char* [lenArray]; //pointer array

     for (int n=0; n < lenArray; n++)//initialize the pairsList array
                                  //pairsList[n]=0;

     for (int i=0; i < num; i++)
     {
       char* nextFreq = new char[MAX_STRING_LENGTH+1]; //frequency array
       char* nextPower = new char[MAX_STRING_LENGTH+1];//amplitude array
       //enter frequency and amplitude pairs i.e 10ghz .1db
       printf("Enter Freq %d: ", i+1);
       scanf("%s", nextFreq);
       printf("Enter Power %d: ",i+1);
       scanf("%s", nextPower);
       pairsList[2*i] = nextFreq;//frequency
       pairsList[2*i+1]=nextPower;//power correction
     }

 unsigned char str[256];//buffer used to hold SCPI command

 //initialize the signal generator's user flatness table
 sprintf((char*)str,":corr:flat:pres\n"); //write to buffer
 viWrite(vi, str,strlen((char*str),0);    //write to PSG
 char c = ',';//comma separator for SCPI command
 for (int j=0; j< num; j++)   //download pairs to the PSG
```

```
      {
       sprintf((char*)str,":corr:flat:pair %s %c %s\n",pairsList[2*j], c,
              pairsList[2*j+1]); // << on SAME line!
       viWrite(vi, str,strlen((char*)str),0);
      }
     //store the downloaded correction pairs to PSG memory
     const char* fileName = "flatCal_data";//user flatness file name
     //write the SCPI command to the buffer str
      sprintf((char*)str, ":corr:flat:store \"%s\"\n", fileName);//write to buffer
     viWrite(vi,str,strlen((char*)str),0);//write the command to the PSG
     printf("\nFlatness Data saved to file : %s\n\n", fileName);

   //load corrections
   sprintf((char*)str,":corr:flat:load \"%s\"\n", fileName); //write to buffer
   viWrite(vi,str,strlen((char*)str),0); //write command to the PSG
   //turn on corrections
   sprintf((char*)str, ":corr on\n");
   viWrite(vi,str,strlen((char*)str),0");
   printf("\nFlatness Corrections Enabled\n\n");
   for (int k=0; k< lenArray; k++)
     {
      delete [] pairsList[k];//free up memory
     }
      delete [] pairsList;//free up memory
    }

    viClose(vi);//close the sessions
    viClose(defaultRM);

    return 0;
}
```

Data Transfer Troubleshooting (E4438C and E8267D Only)

NOTE This section applies only to the E4438C with Option 001, 002, 601, or 602001, 002, 601, or 602, and the E8267D with Option 601 or 602.

This section is divided by the following data transfer methods:

"User File Download Problems" on page 64

"PRAM Download Problems" on page 65

"User FIR Filter Coefficient File Download Problems" on page 66

Each section contains the following troubleshooting information:

- a list of symptoms and possible causes of typical problems encountered while downloading data to the signal generator
- reminders regarding special considerations and file requirements
- tips on creating data, transferring data, data application and memory usage

User File Download Problems

Table 16 Use-File Download Trouble - Symptoms and Causes

Symptom	Possible Cause
At the RF output, some data modulated, some data missing	Data does not completely fill an integer number of timeslots. If a user file fills the data fields of more than one timeslot in a continuously repeating framed transmission, the user file will be restarted after the last timeslot containing completely filled data fields. For example, if the user file contains enough data to fill the data fields of 3.5 timeslots, firmware will load 3 timeslots with data and restart the user file after the third timeslot. The last 0.5 timeslot worth of data will never be modulated.

Data Requirements

- The user file selected must entirely fill the data field of each timeslot.
- The user file must be a multiple of 8 bits, so that it can be represented in ASCII characters.
- Available volatile memory must be large enough to support both the data field bits and the framing bits.

Requirement for Continuous User File Data Transmission

"Integer Number of Timeslots" Requirement for Multiple-Timeslots

If a user file fills the data fields of more than one timeslot in a continuously repeating framed transmission, the user file is restarted after the last timeslot containing completely filled data fields. For example, if the user file contains enough data to fill the data fields of 3.5 timeslots, the firmware

loads 3 timeslots with data and restart the user file after the third timeslot. The last 0.5 timeslot worth of data is never modulated.

To solve this problem, add or subtract bits from the user file until it completely fills an integer number of timeslots

"Multiple-of-8-Bits" Requirement

For downloads to bit and binary memory, user file data must be downloaded in multiples of 8 bits (bytes), since SCPI specifies data in bytes. Therefore, if the original data pattern's length is not a multiple of 8, you need to:

- add bits to complete the ASCII character
- replicate the data pattern to generate a continuously repeating pattern with no discontinuity
- truncate the excess bits

NOTE The "multiple-of-8-bits" data length requirement is in *addition* to the requirement of completely filling the data field of an integer number of timeslots.

Using Externally Generated, Real-Time Data for Large Files

When the data fields must be continuous data streams, and the size of the data exceeds the available PRAM, real-time data and synchronization can be supplied by an external data source to the front-panel DATA, DATA CLOCK, and SYMBOL SYNC connectors. This data can be continuously transmitted, or can be framed by supplying a data-synchronous burst pulse to the EXT1 INPUT connector on the front panel. Additionally, the external data can be multiplexed into internally generated framing

PRAM Download Problems

Table 17 PRAM Download - Symptoms and Causes

Symptom	Possible Cause
The transmitted pattern is interspersed with random, unwanted data.	Pattern reset bit not set. Insure that the pattern reset bit (bit 7, value 128) is set on the last byte of your downloaded data.
ERROR -223, Too much data	PRAM download exceeds the size of PRAM memory. Either use a smaller pattern or get more memory by ordering the appropriate hardware option.

Data Requirements

- The signal generator requires a file with a minimum of 60 bytes
- For every data bit (bit 0), you must provide 7 bits of control information (bits 1-7).

Table 18 PRAM Data Byte

Bit	Function	Value	Comments
0	Data	0/1	This is the data (payload) bit. It is "unspecified" when burst (bit 2) is set to 0.
1	Reserved	0	Always 0
2	Burst	0/1	1 = RF on 0 = RF off For non-bursted, non-TDMA systems, to have a continuous signal, set this bit to 1 for all bytes. For framed data, set this bit to 1 for *on* timeslots and 0 for *off* timeslots.
3	Reserved	0	Always 0
4	Reserved	1	Always 1
5	Reserved	0	Always 0
6	EVENT1 Output	0/1	To have the signal generator output a single pulse at the EVENT 1 connector, set this bit to 1. Use this output for functions such as a triggering external hardware to indicate when the data pattern begins and restarts, or creating a data-synchronous pulse train by toggling this bit in alternate bytes.
7	Pattern Reset	0/1	0 = continue to next sequential memory address. 1 = end of memory and restart memory playback. This bit is set to 0 for all bytes except the last byte of PRAM. To restart the pattern, set the last byte of PRAM to 1.

User FIR Filter Coefficient File Download Problems

Table 19 User FIR File Download Trouble - Symptoms and Causes

Symptom	Possible Cause
ERROR -321, Out of memory	There is not enough memory available for the FIR coefficient file being downloaded. To solve the problem, either reduce the file size of the FIR file or delete unnecessary files from memory.
ERROR -223, Too much data	User FIR filter has too many symbols. Real-Time cannot use a filter that has more than 64 symbols (512 symbols maximum for Arb). You may have specified an incorrect oversample ratio in the filter table editor.

Data Requirements

- Data must be in ASCII format.
- Downloads must be in list format.
- Filters containing more symbols than the hardware allows (64 for real-time and 512 for Arb) will not be selectable for the configuration.

Index

Index

Index

V

W