Single-source tool for VHDL & EPICS development

J. T. Anderson last revised 20230519

Table of Contents

Core Concept

The root concept behind this spreadsheet tool design is that in every FPGA-based development that connects to computer systems – especially modern physics experiments that use EPICS or some similar control and monitoring software – has at the root the concept of a *register.* For purposes of this discussion, a *register* is a uniquely addressable object whose size in bits is that of one atomic transaction at the hardware layer. A *register* is something that is an integer number of bytes in size (8, 16, 24 or 32 bits). *Registers* are always written or read as a whole, but any *register* may conceptually contain *bits* or *bit groups* that have specific functions associated with them.

This map of *registers* that contain *bits* and/or *bit groups* is translated at many different points into other formats. Historically these are all very manual procedures that require much duplicitous effort.

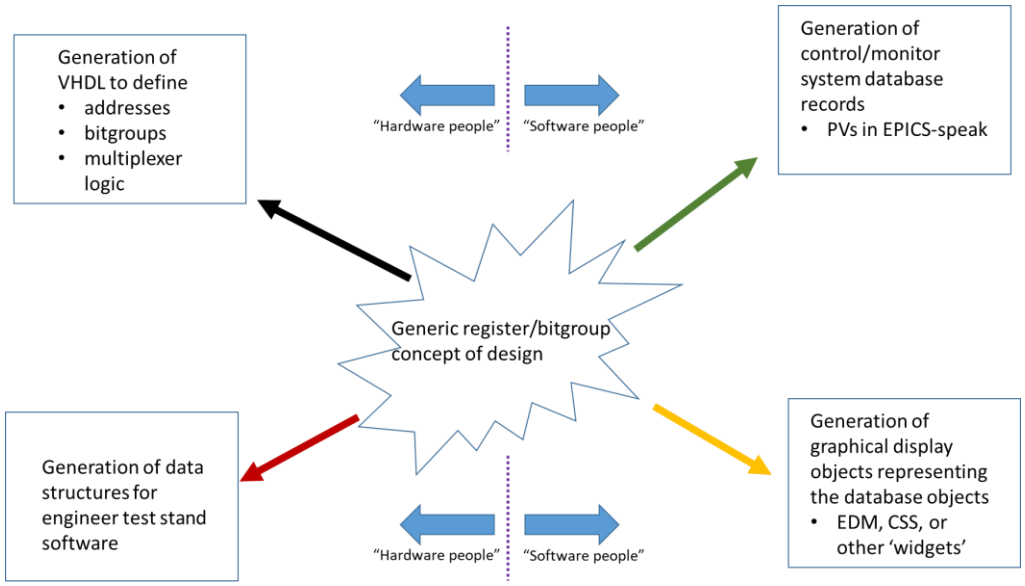


FIGURE 1 - THE SAME MODEL UNDERLIES A MINIMUM OF FOUR DIFFERENT MANUAL DATA ENTRY PROCESSES.

It is altogether too common for firmware to be written, and once debugged by an engineer sufficiently the engineer must then copy/paste/reformat the names of registers and addresses from the firmware into data structures and/or code for the engineering test stand to exercise the FPGA-based hardware in detail. That's one wasteful copy/paste operation. Then once the module is working well enough to try in the end user system, this same information is then re-copied and re-translated *a second time* by a software person into database records. Often, this is even more wasteful of effort on both sides *because the engineer must write a formal document describing the arrangement of registers/bits for the software person to then read and again translate into database language because the software person can't read firmware.* So now there have been at least three copies, if not four of the ***same*** information. Finally, there is yet one more copy/translation that happens between database and graphical display thereof.

Even worse, inevitable updates to firmware arise to fix bugs or add features, and each one of these requires the entire sequence of ***multiple*** copy/translate/paste operations to occur again.

A single source of information is better

"One source to start it all, one place for entry, one click to make them all and files are wrote aplenty....". Apologies to the estate of J.R.R. Tolkien and all fans of the Lord of the Rings books, but this is not completely a jest. The map of registers and bit groups is simply a list. Lists are easy to read and understand, and spreadsheets with code behind them are ideal processors of lists. Just thinking about the way registers are implemented in logic, at

the hardware/firmware level the list starts by having just two columns in a list titled ***Register Name*** and ***Register Address.***

If one analyzes firmware within an FPGA, the firmware implements a list of registers and the most atomic transaction that can occur is a read or write of a register. All registers are the same size, usually an integer multiple of bytes wide. Any given register may implement *bit groups* for selection functions or packing reasons, and there are always single bits that reset or enable logic, but *bit groups* and *bits* rarely can be modified separately from the rest of the register. The vast majority of the time, the concepts of *bits* or *bit groups* are simply read-modify-write transactions of whole registers that software and control system designers try not to think about.

Because of this all software and control system operations all really reduce to reformatting the same information the firmware already has to use into some variant syntax, but the root information that is needed is simply what the registers are and what addresses they exist at in all cases. Then the mental overlay of bits and bitgroups can be applied in the syntax of software and control system languages by simply enumerating all the bits and bitgroups in each register, specifying their bit positions therein. Similarly, at the firmware/hardware level everything is integers. Numerical values the user wants to think about as floating-point numbers never are floating point, there is always some underlying driver or device support software that performs conversion between "user units" and "hardware values".

Spreadsheet-based software overview

Software has been written in Visual Basic for Applications (VBA) that uses as input a Microsoft Excel spreadsheet that enumerates all *registers* within a design and for each register, the *bitgroups* of unique functionality within that register. Primary information such as the address of the register and the span of bits covered by any bitgroup within a register are contained within the spreadsheet. Additional columns of the spreadsheet then contain human readable descriptions, value/meaning associations as simple strings, *directionality* information, initialization values plus whatever else may be needed .

"Languages" generated by the spreadsheet software

The spreadsheet software utilizes a series of global cells for storage of information about the type of processing and the type of system the register map resides in. Up to five different "languages" may be specified for extraction, where each "language" is a reference to a folder of textual ***template files*** that are processed for each line of the spreadsheet to recast the spreadsheet's information into the specify syntax of the languages required. Special features are built into the spreadsheet for VHDL, C and EPICS (Experimental Physics and Industrial Control System) database features, however, the VHDL features are equivalently useful for Verilog, the C features are equivalently useful for Python or Pascal or whatever other language is desired, and the EPICS features are translatable to other control systems.

Language specificity and formatting of output data is managed by the ***template files*** that, line by line, take data from the spreadsheet to perform string substitutions to form output files. At processing end, separate ***assembly files*** are then read by the program to take the various output files and combine them with each other and/or other user-defined files to create final outputs and, if desired, copy/move those final outputs to other folders of the disk.

The software supports template files in up to five different "langugages", where a "language" means a specific folder tree of template files. Within the context of this program the specific language names "VHDL", "C" and "EPICS" are used as stand-ins for the more general terms "firmware language", "software language" and "control system language". A number of optional, advanced spreadsheet functions may use those specific names, but the reader is advised that the spreadsheet system is intended to be more general. Since all files are generated using text substitution controlled by the *template* and *assembly* files there are few places where the exact output language is strictly specified.

Program Phases

The program is invoked by hitting the button *Extract Registers*. When the program runs, it does so in ***phases***, enumerated as Startup, File Open, Template lead-in, Processing, Template lead-out Assembly and File Close. Separate sub-folders of template files, per "language" are processed during Template lead-in, Processing and Template lead-out, generating a variable number of output files. Then during the Assembly phase, the generated output files may be merged with each other and/or external files to form final outputs. During Startup, File Open and File Close no processing of user data, template files or assembly files; these phases are for internal software usage only.

Basic spreadsheet list structure

Realizing that registers are containers of bits, the list then expands such that for every ***Register Name/Address*** row in a list there is a sub-list of ***bit groups*** (single bits or contiguous groups of 'n' bits) that themselves have names, and instead of addresses such bit groups have ***bit ranges.*** These objects all have ***directionality*** (read/write) and obviously all have a ***functional description*** that humans attach to them. At this point the list is relatively easy to picture, and a spreadsheet is an obvious way to capture this data, as shown in Figure 2.

Register/Bitgroup Name	*Address/VHDLBitRange*	*Register/Bitgroup Mode*	*Description*
FPGA_CTL_REG	0x0001	W	
GeC_GainReset	0	W	Hold GeCenter gain analog mux reset when asserted.
BGOsum_GainReset	1	W	Hold BGO sum attenuation analog mux reset
BGOCounterMode	2	W	sets BGO discriminator counter mode
PARSTCounterMode	3	W	sets preamp reset counter mode
PARSTContinuousMode	4	W	sets preamp reset measurment on all the time
PARST_EdgeSel	6..5	W	sets edge recognition for preamp reset monitor
PowerConverterEN	7	W	When high, power board can drive +5V to Raspberry Pi.
Preamp_I2C_OE	8	W	When high, I2C communication with preamp is enabled.
ResetAllScanMachines	9	W	Resets preamp, power board and dongle scan ROM machines
ILAsubSelect	10	W	Sub-selection bit for ILA multiplexers.
PowerBoardI2CFifoReset	11	W	Resets just the command FIFO of the power board I2C machi
PreampI2CFifoReset	12	W	Resets just the command FIFO of the preamp I2C machine
DongleI2CFifoReset	13	W	Resets just the command FIFO of the dongle I2C machine
I2CReset	14	W	Holds preamp, power board and dongle I2C machines all rese
PowerConverterOE	15	W	When high, I2C communcation with power board is enabled.
PA_RESET_COUNT	0x0002	R	current preamp reset count value.
BGOP_MUX_CTL_REG	0x0003	W	
BGOPatternCycleDelay	10..0	W	time between switches of auto-cycling BGO tube mux
BGOPatternFixedValue	13..11	W	Static value for BGO tube mux
BGOPatternMuxMode	14	W	0:count 1:static addresss
	15	X	bit is unused.

FIGURE 2 - SAMPLE LIST OF REGISTERS AND THEIR BIT GROUPS.

Directionality of registers and bit groups

Sometimes a bitgroup is just a single bit, sometimes it is a contiguous range of bits. Some registers may not have bit groups at all but are just a value in and of themselves. The column shown in Figure 2 above labeled ***Register/Bitgroup Mode*** is ***directionality***. There is a limited set of ***directionality*** values, summarized here.

- R (read only of simple register) / FR (read only of FIFO buffer)
- RW (readable and writable, simple register) / FRW (bidirectional FIFO buffer)
- W (write only, single register) / FW (write only FIFO buffer)
- PW (***pulsed*** write, creates internal logic pulse, no register or FIFO that stores value written)
- X to indicate "not used" (for explicit identification of unused bits within a register)

For readers not intimately familiar with hardware, Figure 3 shows how each directionality value works. In that picture, "Fabric Logic" means "the rest of the design", whether implemented discretely or in an FPGA.

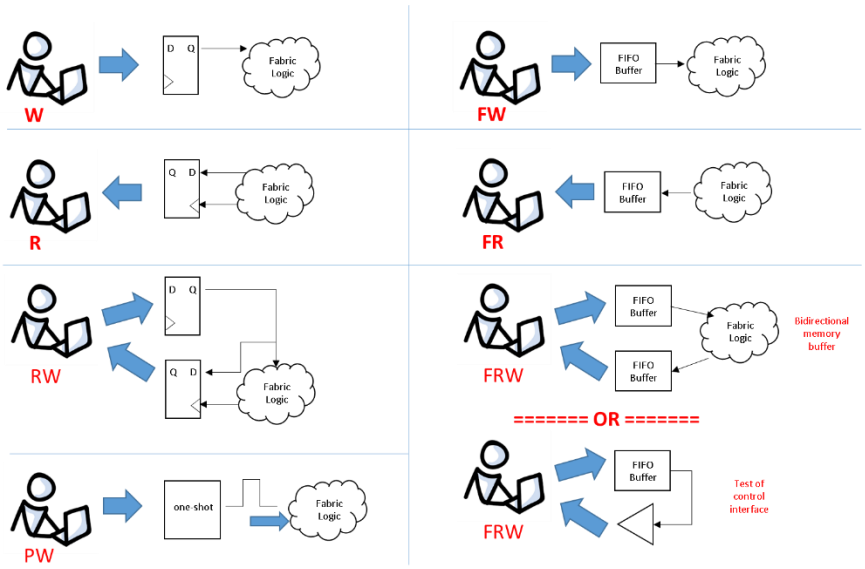


FIGURE 3 - PICTORIAL REPRESENTATION OF DIRECTIONALITY.

Short description of how this software ties to EPICS and other control systems

The EPICS control system views all settable or readable information as ***process variables (PVs)***. Each *process variable* has a value and directionality as defined above, so there is clearly a one-to-one relationship between every ***register*** and/or ***bitgroup*** to a PV. The only difference is that PVs only work in one direction. Hardware objects that are "R", "FR", "W", "PW" or "WO" map to a single PV; hardware objects that are "RW" or "FRW" normally require two. Even the "X" directionality of hardware translates to PVs, but here it may have the meaning of "thing in the hardware that is not exposed to end user" in addition to its previous meaning "unused bits".

Specific to EPICS, the traditional types of PVs are

- ai (analog in) & ao (analog out) – a value with optional scaling and optional user units
  - most commonly, this is used for a ***register***.
  - Some EPICS implementations differentiate between *longin/longout* and *ai/ao*, where *longin/longout* are used for PVs that actually talk to hardware and *ai/ao* are reserved as user-facing PVs that perform scaling between "user units" and associated *longin/longout* PVs that are in "hardware values".
- bi (bit in) & bo (bit out) – single bit values
- mbbi (multibit in) & mbbo (multibit out) – obviously the same as a ***bitgroup.***

Without getting into the details now, it should at this point be clear that EPICS PVs are inherently organized row-by-row the same way that the hardware/firmware is, and that the same concepts of ***register***, ***bitgroup*** and ***directionality*** apply. Yes, there will have to be some additional columns to support things like scaling, user units or subsidiary PV information, but an overall algorithm that simply scans the spreadsheet from beginning row to end row obviously can generate control system databases at the same time that it is generating firmware or software source.

A deeper dive into the design of the spreadsheet

The spreadsheet design uses the top rows to define all the overall settings that control the operation of the process and provide some status to the user. This section of global controls is separated from the user input

area of registers and bit groups by a line that contains the exact string ***#START_OF_DATA*** in column A. All rows after the ***#START_OF_DATA*** row up to and including the row whose column A value is exactly ***#END_OF_DATA*** are processed. For clarity, by convention the #START_OF_DATA and #END_OF_DATA rows are formatted with a red background with bold white text. The magenta button labeled **Set To Standard Colors** will go through the entire spreadsheet from #START_OF_DATA to #END_OF_DATA and set all lines to a specific color scheme based upon the data found in Column A. Use of this button is recommended occasionally to maintain readability.

After entering the data describing the registers and bit groups, ***one*** click of the **Extract Registers** button at the top of the spreadsheet generates all output files. If an error occurs during processing a dialog box will pop up indicating what the error is, what spreadsheet line the error is on, and where possible which column of that line caused the error. For a typical spreadsheet, running **Extract Registers** only takes a few seconds, so cleaning up input errors is easy.

Figure 4 provides a zoomed-out view of the top rows of the spreadsheet (global processing controls). Four areas are highlighted using differently colored rectangles.

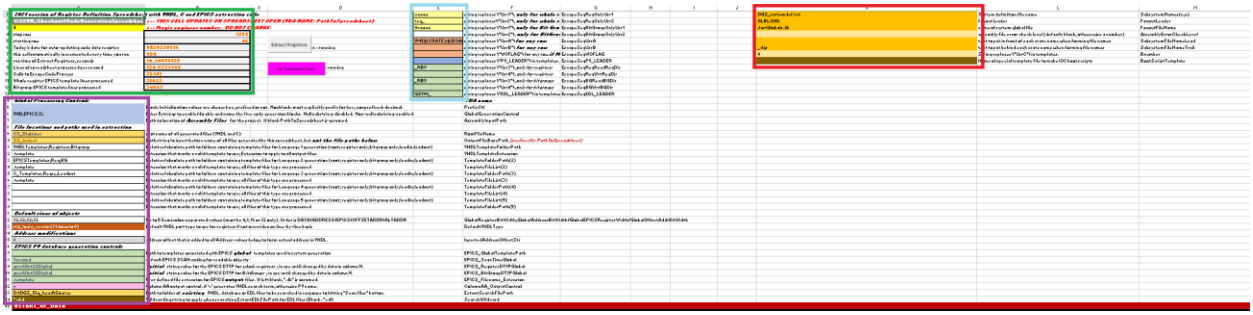


FIGURE 4 - BIRD'S EYE VIEW OF THE GLOBAL CONTROLS OF THE SPREADSHEET

- The ***green*** rectangle indicates *calculated cells* that should not be changed by the user. These cells define the range of rows the extraction software will process and provide a couple auto-updating cells for things like the path to where the spreadsheet is found, the date and a version number that increments every time the extraction code is run. The software uses keywords that the user enters (#START_OF_DATA and #END_OF_DATA) to mark the starting row and ending row to be processed, and the calculation cells highlighted by the green rectangle in Figure 4 auto-update based upon what cells the keywords are found in. The other calculated cells are updated every time the **Extract Registers** button is pressed.
- The ***purple*** rectangle indicates the global controls of the spreadsheet. They are grouped by color into four types of global controls:
    - overall processing flags
    - file location and path controls
    - base names for generated outputs
    - controls specific to VHDL (firmware language) and EPICS (control system language) processes
- The ***blue*** rectangle indicates various user-defined strings that are generally specific to control system database (EPICS) generation functions of the extraction code. These strings are tied to the *Template Files* that define how process variables are generated row-by-row.
- The ***red*** rectangle indicates a 2nd set of user-defined strings specific to control system database (EPICS) generation functions. These strings control the generation of *system databases*, used to develop process variable *fan-out trees* and the number of output files generated. For a singular design that doesn't have any need to make a system of multiple instances of objects with database fan-out controls, this section is not required and should be left blank.

Use of columns within processed rows of spreadsheets

The VBA code ***Extract Registers*** will, when invoked, read every row of the spreadsheet starting with the row following the **#START_OF_DATA** row up to and including the **#END_OF_DATA** row. When the read occurs, columns A through Z of the current row are processed, and data is written to column AA of the row being processed. The following table summarizes how the different columns within the spreadsheet are interpreted, row by row, for the ***normal*** cases of defining registers and bit groups. Reference may be made herein to 'C', 'VHDL' or 'EPICS' as these were the software, firmware and control system languages this software was originally developed for, but reference to these names within this table should be read as example rather than requirement.

Column	***Nominal usage***	***Relates to***
A	Comment or Command. If the cell value starts with '#' then the cell value is compared against the list of known ***commands*** and if a match is found the rest of the rows are interpreted following the rules of the ***command.*** A non-blank cell that does not start with '#' means the entire row is a ***comment*** and none of the data is processed. A ***blank*** cell is interpreted as the command ***#BitGroupDefinition*** as this is the most common data value for column A.	All languages
B	This column contains the *name* of the object being processed.	All languages
C	This column contains the *address* of the register if column A is ***#RegisterDefinition***, otherwise it is the *bit range* of the bit group (either single digit or range m..n) if column A is ***#BitGroupDefinition.*** For other column A command values, this may be an argument to the command.	All languages
D	This column can be used to specify the subdesign port ***type*** of the object if needed. If left blank, all *registers* are assumed to have the default port type specified in cell A43. If left blank, all *bit groups* are assumed to be either **std_logic** (if single bits) or **std_logic_vector(m downto n)** where the span is as defined in column C. If not blank, the value in the cell is used verbatim, overriding the default, for this row only. This is generally only used when complex data structures such as records are defined. The type of the *register* may be defined as a string and then the type may be used within the *bit groups* of the register to create record-format assignments. In this case the type of the *register* is used as a prefix to the type of the *bit group.* If the design structure does not require this, then column D may be used for any other purpose the user desires.	Firmware extraction
E	This column specifies the ***directionality*** of the object (R, W, RW, FR, FW, FRW or X) as described in Figure 3, for purposes of firmware generation. The value of 'X' means "do not generate". The ***directionality*** of a bit group may be a subset of the directionality of the register (e.g., register is RW, bit group may be R or W) but may not be a superset (e.g. register is W, bit group is RW).	Firmware and databases
F	Free-form textual description of what this object is.	All languages
Columns G-I are optional, used for more complex firmware designs.		
G	Initialization value to apply to the firmware object. Nominally the initialization value is only applicable to registers and not to bit groups of registers. Obviously the ability to initialize is a function of the firmware implementation. If the design structure does not require this, then column G may be used for any other purpose the user desires.	Firmware
H	In cases where the firmware port type (column D) is explicitly specified, the value here may be additionally used to create a port sub-type for use with record data types. The intended usage is as a suffix for use within *bit groups* to allow two layers of records (e.g., <regtype>.<bitgrouptype>.<bitgroupsubtype>). If the design structure does not require this, then column H may be used for any other purpose the user desires.	Firmware
I	Column I is nominally used if the firmware design implements a multiplexer structure for all readable registers, allowing the user to specify which multiplex group the register is in. If the underlying firmware structure does not support or require this, then column I may be used for any other purpose the user desires.	Firmware
Columns J – P are optional, used only if control system databases are being generated.		
J	Nominally this column contains a letter code corresponding to the type of process variable (PV) that is to be generated (X:none, A:"ai" or "ao", B: "bi" or "bo", M: "mbbi" or "mbbo", L: "longin" or "longout"). The distinction between 'i' and 'o', or 'in' vs. 'out', comes from the *directionality* of the register or bit group (column E).	Databases
K	This column nominally contains a string formatted as a list of selections associated with values to set the prompts associated with bit ("bi" or "bo") or multibit ("mbbi" or "mbbo") process variables. The format of these strings will be provided later within this document.	Databases
L	Nominally this column contains a string formatted as a list of process variable field names associated with values to set scaling and alarm factors associated with analog ("ai" or "ao") process variables; the format will be provided later within this document.	Databases
M	This column may be used to provide the process variable in a generated database with a different name than the object name (column B) used by firmware and software.	Databases

N	This column may be used to override the default EPICS "DTYP" and/or "SCAN" field values provided in the global cells for this row only.	Databases
O	If not empty the string contained within this column may be converted within a template file to a multi-line output value. The conversion that is applied is • The character pair '\t' is converted to a tab character in the output • The character pair '\n' is converted to a carriage return/line feed (ASCII code 13, then ASCII code 10) in the output • The character pair '\r' is converted to just a carriage return (ASCII code 13)	Databases
Columns P-Z have a few specific escape code ***uses but are generally freeform for use as desired***		
P	Some template file substitution keys reference this column to provide specialized formatting but in general it has no specifically defined use.	Any
Q	Some template file substitution keys reference this column to provide specialized formatting but in general it has no specifically defined use.	Any
R	Some template file substitution keys reference this column to provide specialized formatting but in general it has no specifically defined use.	Any
S	Some template file substitution keys reference this column to provide specialized formatting but in general it has no specifically defined use.	Any
T	Some template file substitution keys reference this column to provide specialized formatting but in general it has no specifically defined use.	Any
U	No special usage. Available as data to template files through keyword substitution.	Any
V	No special usage. Available as data to template files through keyword substitution.	Any
W	No special usage. Available as data to template files through keyword substitution.	Any
X	No special usage. Available as data to template files through keyword substitution.	Any
Y	No special usage. Available as data to template files through keyword substitution.	Any
Z	No special usage. Available as data to template files through keyword substitution.	Any

TABLE 1 - SUMMARY USAGE OF DIFFERENT COLUMNS WITHIN THE SPREADSHEET

Templating and Template Files

The term ***template***, or ***template files***, will occur throughout this document. What this specifically refers to is the way that the spreadsheet forms the information in its many output files. All forms of output – software, firmware, database, GUI, etc. are formed using ***template files.*** A template file is simply text that describes the overall syntax in the language of choice for access to a register value. The majority of this text will be the same for all registers and bitgroups with only a few specific characters that must vary for each instance.

The spreadsheet data in the rows and columns is the variant data per object, so all the software of the spreadsheet must do is have a way to perform string substitutions within these templates. This is done by inserting specially formatted strings referred to as ***Escape Codes*** within each template file. ***Escape Codes*** are simply specially formatted text that that is highly unlikely to occur normally, that can be used as a lookup for a string related to the spreadsheet line being processed.

For example, a C program may require a **struct {}** enumerating the addresses of all the registers in the design. Such a **struct {}** is easy to make, it requires only three parts:

1. A line of the form **typedef struct {**
2. A line for every register creating a structure element for that register
3. A closing line of the form **} STRUCTNAME;**

A simplistic template file may then be created with a single line of the form to do step 2 of the list above, for every register that may be defined within the spreadsheet:

```
unsigned short int \*CurrentRegister*\\*;*\  //Addr: \*Address*\ Function: \*RegisterFunction*\
```

FIGURE 5 - EXAMPLE LINE FROM TEMPLATE FILE SHOWING ESCAPE CODES

Relationship between template files, "languages", and output files

As each line is processed, ***all*** template files of the specified type (Register, Bitgroup, Lead-in, Lead-out) in ***all*** specified "language" folders (from cells A21-A30) are processed. Each template file creates its own output file.

As a large number of output files may be created, an output subfolder is highly recommended. See *File and Path Information* on *page 12*.

There is a convention for the naming of the output files generated by templates. The convention is that

a) The name of the output file will be the same as the root name (sans file extension) of the template file
b) The *file extension* of the output file will be created based upon the name of the "language" of the template file (the substring from cell A16) and the *type* of the template file (Register:"_REG", Bitgroup:"BG", Lead-in:"_LI", Lead-out:"_LO").

This will result in unique names for every output file even if two template files in two folders have the same name. As an example, a template file named "XYZ.template" in the Registers sub-folder of the VHDL language set will generate an output file named "XYZ.VHDL_REG".

Comments and plaintext within template files

Template file lines with no *escape codes* within a template file are simply copied verbatim to the output, unless the line's first two characters are "\\". This character pair – that must be the first two characters of the line – marks the line as a *comment* that is skipped without being processed or copied in any way.

What defines an escape code?

This template file example above contains four ***escape codes***, namely *CurrentRegister*\, *;*\, *Address*\ and *RegisterFunction*\. An ***escape code*** is defined as the text *between* the special character pairs ***** and ***** . Template files are processed line by line. For each line when * is found in a line of a template file additional characters are read until the terminating character pair ***** is read. Failure to match each * with a following *\ will cause errors. The text *between* ***** and ***** is then compared against a list of known keywords with case insensitive matching, and if matched the entire escape code (including the * and *\) is stripped ***and replaced by data from the current line of the spreadsheet*** or in some cases the result of simple text formulas based upon other spreadsheet conditions.

Output from the example template file line

A spreadsheet defining a few registers would then generate an output file that reads something like

```
unsigned short int PULSED_CONTROL_REG; //Addr: 0 Function: Pulsed (write-only) controls to FPGA
unsigned short int SERDES_CTL_REG; //Addr: 1 Function: FPGA_CTL_REG readback
unsigned short int LED_REG;   //Addr: 2 Function: For LED control on board
unsigned short int DIAGNOSTIC_CTL_REG; //Addr: 3 Function: As named
unsigned short int TIMESTAMP_HIGH;    //Addr: 6 Function: Latched on plsd ctl or trig
unsigned short int TIMESTAMP_OFFSET;  //Addr: 7 Function: Currently this register is unused.
unsigned short int ACCEPT_MSG_DELAY_REG; //Addr: 8   Function: Provides delay before assertion
unsigned short int CODE_DATE_REG;     //Addr: 126    Function: firmware date
unsigned short int CODE_REVISION_REG  //Addr: 127    Function: firmware revision
```

by processing the one-line template for every #RegisterDefinition line in the spreadsheet.

Many predefined *escape codes* are available and are summarized in the appendices of this document. Every cell from column A to column Z of the line being processed is made available for use in template files as direct string substitution by using *escape codes* *A*\ through *Z*\.

Assembly files complement Template files

After processing all lines of the spreadsheet, the program will then process ***Assembly Files.*** The differentiation between a ***Template*** file and an ***Assembly*** file is that ***template*** files are used line by line as the spreadsheet processes the input data to generate formatted outputs related to all the registers and bit groups, but ***assembly*** files explain *how to build the final output **after** the spreadsheet runs.* Any boilerplate text such as the line of the form **typedef struct {** or the closing line of the form **} STRUCTNAME;** may be generated by using the

Assembly file to write "lead-in" boilerplate to the final output, then copy the file generated from templates to the final output, then write "lead-out" boilerplate to the final output and close the final output file.

Template File Sections

There are cases in which a single template for all outputs of a given type is insufficient. For example, VHDL assignment statements within the generated register subdesign may have variations in format based upon whether a register and/or bit group within a register are part of a VHDL record structure, part of an array of registers, or a singular object. As another example a user may wish to generate a secondary C data structure file in addition to the one that enumerates all defined registers that only enumerates *some* registers that happen to be associated with a specific part of the hardware.

The VBA code supports this by the use of *Section Headers* within template files. A *section header* is formatted similarly to an *escape code*, but with different characters. A section header begins with the character pair \! and ends with the character pair !\ and ***must*** be followed on the same line by at least one space and then an *escape code* (character pairs * and *\ with text between). Section header lines themselves never are copied to the output but are used to potentially change the state of a flag variable within the VBA code that enables/disables processing of lines within the template file or causes processing of the template file to terminate before end-of-file. This allows any given template file to have multiple templates within it but to control which (if any) template section is used based upon spreadsheet data.

When a section header is seen, the *escape code* that follows it is processed by the *Escape Code Processor* and the resulting string returned by the Escape Code Processor is compared (case insensitive) to the text between the \! and !\ character pairs of the *Section Header.* If the string returned by Escape Code Processor matches the text of the Section header, the flag variable is set to enable processing, otherwise the flag is set to disable processing. Many combinations of *section headers* can be made to develop whatever logic is needed within a template file, and these are described in detail in *Appendix 4: Details of Section Header logic* on *page 26*.

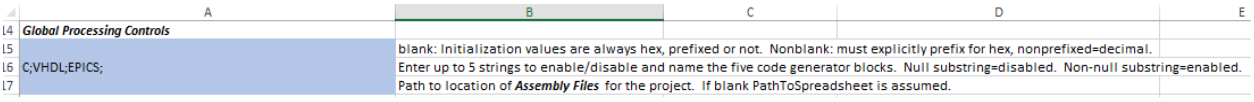
Enough with concepts, how do I use this?

The first step is to put overall settings of the program into the global controls of the spreadsheet. After overall settings are set, the user then enters data describing the registers, one per line. After all registers have been entered, the user then inserts lines *after* each register defining the bit groups contained within the register. Once all registers and bit groups have been defined, the user then clicks the ***Extract Registers*** button at the top of the spreadsheet to run the code.

Global controls of spreadsheet

The first set of control strings are the most generic; these are shown in Figure 6.

- The first controls how numbers are processed for register initialization values.
- The second controls ***the number of different "language template sets"*** that will be defined in cells to follow and gives them names. The names have some meaning here in that some spreadsheet commands *only* operate on the 'C' template set (e.g., #ArrayExtractStart) and some spreadsheet commands *only* operate on the 'VHDL' template set (e.g. #EntityDeclarationStart).
- The third defines the folder location of the ***assembly files*** for the project, if they are not in the same folder as the spreadsheet.



	A	B	C	D	E
14	***Global Processing Controls***				
15		blank: Initialization values are always hex, prefixed or not. Nonblank: must explicitly prefix for hex, nonprefixed=decimal.			
16	C;VHDL;EPICS;	Enter up to 5 strings to enable/disable and name the five code generator blocks. Null substring=disabled. Non-null substring=enabled.			
17		Path to location of ***Assembly Files*** for the project. If blank PathToSpreadsheet is assumed.			

FIGURE 6 - OVERALL PROCESSING (MOST GLOBAL) CONTROLS

First time user instructions for cells A15-A17

If you're just starting out, leave cells A15 and A17 blank and enter the string **C;** into cell A16. This tells the spreadsheet to just process 'C' (software) templates.

File and Path Information

The second set of control strings define names and paths of all files referenced or generated by the spreadsheet. Each is explained tersely by the text to the right of the global cell. See Figure 7.

	A	B
14	***Global Processing Controls***	
15		blank: Initialization values are always hex, prefixed or not. Nonblank: must explicitly prefix for hex, nonprefixed=decimal.
16	C;VHDL;EPICS;	Max 5 semicolon-separated strings naming 'language' of template folderd defined in A21-A30.
17		Path to location of ***Assembly Files*** for the project. If blank PathToSpreadsheet is assumed.
18	***File locations and paths used in extraction***	
19	XIA	root name of all generated files (VHDL and C)
20	SS_output	Path string to insert before name of all files generated by this spreadsheet, but ***not the file paths below***
21	C_Templates;Regs;;;Leadout	Rel/abs path to language #1 definition folders (root; register only;bitgroup only;leadin;leadout)
22	.template	Extension that marks a valid template to use; Extension to apply to all output files.
23	VHDLTemplates;Registers;;;;	Rel/abs path to language #2 definition folders (root; register only;bitgroup only;leadin;leadout)
24	.template	Extension that marks a valid template to use; all files of this type are processed
25	EPICSTemplates;Reg;BG;;Leadout	Rel/abs path to language #3 definition folders (root; register only;bitgroup only;leadin;leadout)
26	.template	Extension that marks a valid template to use; all files of this type are processed
27		Rel/abs path to language #4 definition folders (root; register only;bitgroup only;leadin;leadout)
28		Extension that marks a valid template to use; all files of this type are processed
29		Rel/abs path to language #5 definition folders (root; register only;bitgroup only;leadin;leadout)
30		Extension that marks a valid template to use; all files of this type are processed

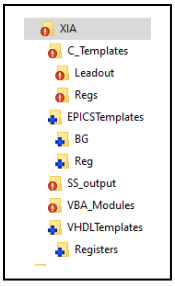


FIGURE 7 - EXAMPLE DEFINITION OF "LANGUAGES" IN A CODE GENERATING SPREADSHEET AND ASSOCIATED FILE STRUCTURE

Cells A19 and A20 control the naming and location of generated output files. Cells A21-A30 are treated as five ***language setting pairs*** following the order of names in the string of cell A16. For the example of this document, cell A16 specifies that three of the possible five language sets are to be used, named 'C', 'VHDL' and 'EPICS' respectively. This means that cell A21 will define the location and folder substructure of the template files for the 'C' language and that cell A22 will define the file extension that defines a file to be a template file within the folder structure defined in cell A21. Similarly cells A23/A24 are the 'VHDL' set and cells A25/A26 are the 'EPICS' set.

First time user instructions for cells A21-A30

For a first-time start, put the string **C_Templates;Regs;;;;** in cell A21 and the string **.template** in cell A22. Leave cells A23 through A30 blank. This will process only templates for ***#RegisterDefinition*** lines.

Global cells specific to firmware generation (VHDL)

The next section of global data defines items specific to VHDL generation, as shown in Figure 8.

	A	B	C	D
31	***Default sizes of objects***			
32	16;16;16;16;	Up to 5 Semicolon-separated values (must be 4,8,16 or 32 only). Order is DATA/ADDRESS/EPICS/OFFSETADDR/ALTADDR		
33	std_logic_vector(15 downto 0)	Default VHDL port type to use for registers if not overriden on line-by-line basis		
34	***Address modifications***			
35	0	address offset that is added to all Address values below to form actual address in VHDL		

FIGURE 8 - VHDL GLOBAL CONTROLS.

Cell A32 specifies the size, in bits, of values to be generated by processing *escape codes*. This normally matches the default port type for registers defined in cell A33, which presumably is also the width of data words used in the communication interface. Addresses, however, are often a different size (in bits) than the data so this semicolon-separated string allows for them to be different sizes. The spreadsheet generates hexadecimal values in many *escape code* strings, so the bit-width must be an integer multiple of four.

Cell 35 provides a place to enter an *address offset* that may be added to the address given for any register prior when processing *escape codes*. For instance, there may be only 256 registers within the design, but the physical address might be from 0x10000 to 0x100FF. VME systems often are like this.

First time user instructions for cells A31-A35

For a first-time start, fill cells A32, A33 and A34 with exactly what's in Figure 8.

EPICS global settings

Figure 9 defines the global settings for EPICS (control system database) generation. Some of these cells are default values that will be used for all lines unless specifically overridden by entering data in one of the later columns (e.g., columns M through P). Others control specifics of process variable (PV) generation or modify output file names.

	A	B	C	D	E	F
36	***EPICS PV database generation controls***					
37		Path to templates associated with EPICS ***global*** templates used in system generation				EPICS_GlobalTemplatePath
38	1 second	default EPICS SCAN coding for readable objects				EPICS_ScanTimeGlobal
39	asynUInt32Digital	***Initial*** string value for the EPICS DTYP for *whole registers* ; in use until changed by data in column N.				EPICS_RegisterDTYPGlobal
40	asynUInt32Digital	***Initial*** string value for the EPICS DTYP for *BitGroups* ; in use until changed by data in column N.				EPICS_BitGroupDTYPGlobal
41	.template	User defined file extension for EPICS **output** files. If left blank, ".db" is assumed.				EPICS_Filename_Extension

FIGURE 9 - EPICS GLOBAL CONTROLS.

First time user instructions for cells A37-A41

For a first-time start, since you're not generating databases, leave cells A37-A41 blank.

Auxiliary functionality

In addition to file generation, a simplistic "grep"-like tool is built into the spreadsheet program to ease maintenance purposes. When the spreadsheet is run, the name of the object or the PV name associated with the object is written to column AA, line by line. The purpose of this is for later use, if desired, by the file search functionality. The relevant control cells for this are A42 through A44, but as this is an auxiliary function explanation of this function shall be deferred to an appendix.

	A	B	C	D
42	v	Column AA output control. if 'v' generates VHDL search term, otherwise PV name.		
43		Path to folder of ***existing*** VHDL database or EDL files to be searched in response to hitting "Scan files" button.		
44		Wildcarding string to apply when searching ExtantEDLFilePath for EDL files (Blank : *edl)		

FIGURE 10 - AUXILIARY FILE SEARCH CONTROL CELLS

First time user instructions for cells A42-A44

For a first-time start leave cells A42-A44 blank.

Entering data for registers and bit groups

The spreadsheet uses column A as the ***Command*** cell for every row. The user must enter the exact string **#START_OF_DATA** in column A of the first row before user data. The user must enter the exact string **#END_OF_DATA** in column A at the row after the last row of user data. The spreadsheet looks for these values and puts the row number at which processing will start in cell B5; the row at which processing will stop is put in cell B4. These are both in the ***calculated cells*** section of the spreadsheet outlined in the ***green*** rectangle of Figure 4.

The user enters data in the rows between but not including the rows labeled **#START_OF_DATA** and **#END_OF_DATA**. The data in column A tells the spreadsheet how to handle each row. Many ***commands*** are defined for column A. Registers are defined on lines whose column A data is **#RegisterDefinition.** The command for bit groups is **#BitGroupDefinition**, although since this is the most common line type, a blank column A is treated as if column A contained **#BitGroupDefinition**.

All defined commands begin with the hash character '#', and ***the commands are case-sensitive***. Any line whose column A is *non-blank* but not recognized as a known keyword *is treated as all comment.* This allows the user to put any comments desired in line with definitions and can be very useful – such as a line that contains text in each cell to remind you what each column does, repeated every 50 lines or so.

First time users: define a register

To define a register, perform the following steps.

1. Enter the exact string **#RegisterDefinition** into column A. This tells the spreadsheet that this line defines a register.
2. Enter the **name** of the register into column B.
3. Enter the **address** of the register into column C.
4. Enter the **directionality** of the register into column E (i.e., R, RW, W, etc.).
5. Type in any textual **description** desired into column F.
6. For now, ignore columns to the right of F.

When done you should have something like Figure 11 below.

	A	B	C	D	E	F
95	*Comment or Control String*	*Register/Field Name*	*Address/VHDLBitRange*	*VHDLType/2nd Mailbox*	*Register/Field Mode*	*Description*
96	#RegisterDefinition	PROPAGATION_CONTROL_REG	12		RW	controls TTCL decode

FIGURE 11 - EXAMPLE OF A REGISTER DEFINITION LINE

First time users: define bit groups of a register

Bit groups are always listed immediately ***after*** the line defining the register they are part of. All bit groups for a register must be defined before the next register is defined. A bit group definition has the following entries; see example in Figure 12.

1. Column A may be filled with the exact string **#BitGroupDefinition** but may be left blank, as the program assumes that any line with a blank column A is a #BitGroupDefinition.
2. Enter the **name** of the bit group into column B.
3. Enter the **bit range** of the bit group into column C. A **bit range** is either just a single number or a contiguous bit span.
    a. *Contiguous* bit spans are defined by two numbers with two periods between them, such as 13..5, analogous to VHDL syntax of (13 downto 5).
        i. The first number must always be larger than the 2nd number.
4. Enter the **directionality** of the bit group into column E (i.e. R, RW, W, etc.), just like was done for a register.
    a. The **directionality** of a bit group must be the same as or a logical subset of the **directionality** of the register. You can't have a register with directionality of "R" (read only) and have a bit group with directionality "RW" (read/write) within that register.
5. Type in any textual **description** desired into column F.

	A	B	C	D	E	F
95	*Comment or Control String*	*Register/Field Name*	*Address/VHDLBitRange*	*VHDLType/2nd Mailbox*	*Register/Field Mode*	*Description*
96	#RegisterDefinition	PROPAGATION_CONTROL_REG	12		RW	controls TTCL decode
97		Prop_F1	0		RW	propagate T3 and sync
98		Prop_Trig_Decision1	1		RW	enable decode of frame 3
99		Prop_Trig_Decision2	2		RW	enable decode of frame 4
100		Prop_Trig_Decision3	3		RW	enable decode of frame 5
101		Prop_Trig_Decision4	4		RW	enable decode of frame 6
102		Prop_Trig_Decision5	5		RW	enable decode of frame 7
103		Prop_Trig_Decision6	6		RW	enable decode of frame 8
104		Prop_Trig_Decision7	7		RW	enable decode of frame 9
105		Prop_Trig_Decision8	8		RW	enbl decode of frame 10
106		Prop_F12	9		RW	propagate F12 commands
107		Prop_F14	10		RW	propagate F14 commands
108		unused	11		RW	
109		Prop_F16	12		RW	propagate F16 commands
110		Prop_F17	13		RW	propagate F17 commands
111		unused	15..14		RW	

FIGURE 12 - A REGISTER DEFINITION WITH BIT GROUPS.

Ending the definition of a register

After entering all the bit groups the user may explicitly terminate the register definition by adding a line with the explicit command **#EndRegisterDefinition,** but this is rarely needed. A subsequent **#RegisterDefinition** line or the **#END_OF_DATA** will implicitly end the previous **#RegisterDefinition** and cause the software to process the template files for the previous **#RegisterDefinition.** This notion of "closure" is necessary because some facets of generating outputs for the whole register (such as read or write masks in firmware) may be dependent upon which bits are populated or not within the register by the bitgroups. To handle this the software sets a "register in

progress" flag bit when it encounters the **#RegisterDefinition** line but doesn't actually process the template files for the whole register until the register definition is complete.

#EndRegisterDefinition is only required if the next line after the end of a register's bitgroups is some special command such as **#EntityDeclarationStart, #ArrayExtractStart** or one of the **#EPICS…** commands.

Running the code

Assuming all global setup is complete, the code is run by hitting the *Extract Registers* button at the top of the spreadsheet, or by manually invoking the *Extract_Registers* macro. When the macro is invoked, lines are processed in order. When the routine is complete, a little dialog box stating "Extraction Complete" will pop up.

As lines are processed, errors may cause a message box to pop up explaining the error, such as the example shown in Figure 13. Upon responding with "OK", the program terminates. Fix the error and run ***Extract Registers*** again.

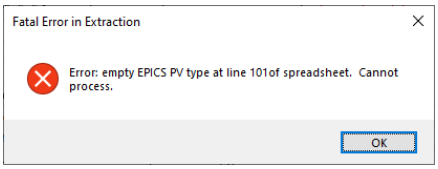


FIGURE 13 - MESSAGE BOX FOR SPREADSHEET DETECTED USER ERROR.

Of course, the spreadsheet can't catch everything. Some kinds of errors may crash the code. If that occurs the spreadsheet will throw a Visual Basic for Applications error message box onto the screen and offer the opportunity to "end" or "debug" the code. Please bring these to the attention of the developer.

Examine the output

After running *Extract Registers* the spreadsheet will have generated files in the folder defined by cell A20. Look in that folder for the output files and examine what has been created.

END OF INTRODUCTION

This concludes the introduction to the basic design of the code generating spreadsheet. Appendices follow.

Appendix 1: Detailed list of column A commands

Commands and file names enumerated in this table are case sensitive and must be spelled precisely as shown here. Anything not matching exactly is assumed to be a user comment and will generate no outputs to files.

The following table lists all the supported command strings for use in column A of the spreadsheet and provides a summary description of what the command will do. Separate appendices are provided that go into detail defining all special commands related to firmware (VHDL), database (EPICS) and software (C) templating.

Command	Definition
Basic commands for register and bitgroup definition	
#RegisterDefinition	States that a new register definition is starting. Causes previous #RegisterDefinition to be marked as complete and thus processes all template files in the **Register** sub-folder of ***all*** defined languages.
#BitGroupDefinition, or a *blank* column A	Defines a bit group within a register. Causes immediate processing of all template files in the **Bitgroup** sub-folder of ***all*** defined languages.
#EndRegisterDefinition	Marks the end of a #RegisterDefinition. Causes previous #RegisterDefinition to be marked as complete and thus processes all template files in the **Register** sub-folder of ***all*** defined languages. Not often needed as this function is implied by a subsequent #RegisterDefinition or seeing #END_OF_DATA.
Firmware creation (VHDL) commands	
#LibraryDefinition or #LibraryUseCase	Invokes processing of a special template file named ***LibraryDefinition.template*** that must be located in the template folder associated with the language name "VHDL". Use of *section headers* within this template file to separate the templating for the two different commands is presumed. In most cases these commands are never used as it is simpler to maintain a text file of standard definitions and then just copy that into generated files using ***Assembly File*** commands, but these lines can find occasional use forming documentation of library dependence within the spreadsheet.
#RecordDefinition, #FieldDefinition, or #RecordEndDefinition	Invokes processing of a special template file named ***RecordDefinition.template*** that must be located in the template folder associated with the language name "VHDL". Use of *section headers* within this template file to separate the templating for the different commands is presumed. Similar to the #Library commands above it is often easier to just have a project-wide text file of such things, but experience has shown that having a reference within the spreadsheet can make defining ports, registers and bitgroups using custom types easier.
#ArrayTypedef	Invokes processing of a special template file named ***ArrayTypedef.template*** that must be located in the template folder associated with the language name "VHDL". Rarely used.
#SignalDefinition	Invokes processing of a special template file named ***SignalDefinition.template*** that must be located in the template folder associated with the language name "VHDL". Rarely used.
#VHDL_Comment	Invokes processing of a special template file named ***Comment.template*** that must be located in the template folder associated with the language name "VHDL". Rarely used but can find occasional use to put separators between blocks of generated lines in comment form.
#EntityDeclarationStart or #EntityDeclarationEnd	Invokes template file processing in ***only*** the *Registers* sub-folder of the template folder tree associated with the language name "VHDL". *Section Headers* within those template files are expected to differentiate response based upon seeing the specific command by comparison against the *escape code* *A*\. **#EntityDeclarationStart** sets an internal flag variable that is not cleared until **#EntityDeclarationEnd** is seen, plus a count of registers processed within the "entity" declaration block is kept. These are both available through *escape codes* to allow the template files to use them as needed.
EPICS commands	
#EPICSCalcout	Invokes processing of a special template file named ***CALCOUT.template*** that must be located in the template folder associated with the language name "EPICS" to build process variables of the "calcout" type whose format differs from other types of process variables.
#EPICSBitgroupGlobal or #EPICSBitgroupLocal	#EPICSBitgroupGlobal invokes processing of a special template file named ***BGGlobal.template*** that must be located in the template folder associated with the language name "EPICS". Similarly, #EPICSBitgroupLocal uses a template file named ***BGLocal.template.***

	These are used to make special-case process variables as described in Appendix 7: Details of specialized control system (EPICS) language support.
#EPICS_FANOUT	Similar in function to ***#EPICSBitgroupGlobal***, this keyword provides a mechanism to create a fan-out tree for devices that provide multiple channels. Used in concert with ***#EPICSBitgroupGlobal*** to make a system fan-out tree allowing the user to set all channels of all modules controlled by a single IOC to a desired value. This command does not use any template file, the structure of output is fixed by the code as described in Appendix 7: Details of specialized control system (EPICS) language support.
C commands	
#ArrayExtractStart or #ArrayExtractEnd	Invokes template file processing in ***only*** the *Registers* sub-folder of the template folder tree associated with the language name "C". *Section Headers* within those template files are expected to differentiate response based upon seeing the specific command by comparison against the *escape code* *A*\.
#C_Comment	Invokes processing of a special template file named ***Comment.template*** that must be located in the template folder associated with the language name "C". Rarely used but can find occasional use to put separators between blocks of generated lines in comment form.
Miscellaneous commands	
#Breakpoint	Can be used in concert with setting a breakpoint on a specific line of the top-level VBA function to break processing on the current spreadsheet line. Intended for use in debugging the VBA code.
#SetBaseOffset	Uses the value in column B as a number (leading 0x = a *hex* number) and changes the value of global variable **InsertedAddressOffset** to that value, overwriting the value derived at program start from cell A35. Provided as support for development of firmware for FPGAs with partitioned address ranges.
#START_OF_DATA	Defines the start of the data extraction range.
#END_OF_DATA	Defines the end of the data extraction range. Causes previous #RegisterDefinition to be marked as complete and thus processes all template files in the **Register** sub-folder of ***all*** defined languages.

TABLE 2 - LIST OF COLUMN A COMMANDS

Appendix 2: Detailed list of supported Escape Codes

The VBA code converts all escape sequences to all uppercase upon reading them, so case is a convenience here, not a necessity. The table below lists all defined *escape codes* as of May 17, 2023.

General Purpose Escape Codes related to spreadsheet cells of the current row	
column ?\, or just *?*\	With "?" being any character from a to z (or A to Z, as case doesn't matter), means "copy the cell of the referenced column in the current row being processed to the output".
?_REG\	With "?" being any character in the set {H,I,J,K,Q,R,S,T,U,V,W,X,Y,Z}, this is the string value of the specified cell from the most recent **#RegisterDefinition** line.
?_REG0\	Same as *?_REG*\ except that if the cell is blank, returns the character "0" instead of null.
Prev_Z\	Similar to *Z_REG*\, this is updated by any **#RegisterDefinition, #EndRegisterDefinition** or the **#END_OF_DATA** line, and contains the value that *Z_REG*\ *would have had* before being updated by the current line. In other words, a pipelined copy of *Z_REG*\. Rarely used.
?0\	With "?" being any character in the set {Q,R,S,T,U,V,W,X,Y,Z}, this is the string value of the specified cell if the cell is not empty, otherwise the character "0".
Expanded_B\ *Expanded_C*\ *Expanded_D*\	The value of the referenced cell but if the cell value contains the characters "\n", "\r" or "\t" these are replaced within the cell value by the newline character, the carriage return character or the tab character. Intended mostly for use with **#VHDL_Comment** or **#C_Comment** commands.
General Purpose Escape Codes related directly to spreadsheet global control cells A18 – A44	
RootFileName\	Value of global control cell A18.
RootFileName\	The value of global cell A19.
Address_Offset\	The value in spreadsheet global cell A35.
General Purpose Escape Codes related to spreadsheet global control cells E1 – E13	
USR1\	If processing a register, whatever the user has put in spreadsheet global cell E1. If processing a bit group, whatever the user has put in spreadsheet global cell E2.
USR2\	If processing a register, whatever the user has put in spreadsheet global cell E3. If processing a bit group, whatever the user has put in spreadsheet global cell E4.
USRA\, *USRB*\	The strings found in spreadsheet global cell E5 or E6, respectively.
WOFLAG\	The data in cell E7 of the spreadsheet ***only*** if the **mode** of the register or field being processed is PW ("pulsed write", or "write only"). Otherwise, the *WOFLAG*\ is stripped with no text added to the output.
PV_LEADER\	Replaced by whatever the user has put in spreadsheet global cell E8. Nominally intended for use specifically in the generation of control system database files.
REGDIR\	Replaced by whatever the user has put in spreadsheet global cell E9, if the PV is of an "input" type; replaced by whatever the user has put in spreadsheet global cell E10 if the PV is of an "output" type.
BGDIR\	Replaced by whatever the user has put in spreadsheet global cell E11, if the PV is of an "input" type; replaced by whatever the user has put in spreadsheet global cell E12 if the PV is of an "output" type.
EDL_LEADER\	Replaced by whatever the user has put in spreadsheet global cell E13. Nominally intended for use specifically in the generation of control system GUI definition files.
General Purpose Escape Codes related to spreadsheet global control cells K1 – K8	
USRc\	The string found in spreadsheet global cell K7, if K7 is not blank.
General Purpose Escape Codes related to operation of the program	
Program_Phase\	Returns a string indicative of the phase of program operation to allow for smarter use of Section Headers.
CurrentTemplateBaseName\	The name of the template file currently being processed. Used only for debugging.
Ssrow\ or *INROW_ABSOLUTE*\ or *INROW*\	The actual row of the spreadsheet being processed.
INROW_RELATIVE\	The row number currently being processed, with the first row after the keyword ***#START_OF_DATA*** defined as row number 1.
INROWX10\	The value for *INROW*\, multiplied by 10. Typically, only used for automated GUI file generation.
RegisterNumber\ *RegNumMod16*\	The current count of how many ***#RegisterDefinition*** lines have been processed. *RegNumMod16*\ is *RegisterNumber*\ modulo 16.

Escape Codes related to general processing of register and bitgroup lines of the spreadsheet	
CurrentRegister\ or *RegisterName*\	The name of the register (column B) from the last ***#RegisterDefinition*** line.
BitgroupName\	The name of the bitgroup (column B) from the last ***#BitGroupDefinition*** line.
Address\	The register address from the last ***#RegisterDefinition*** line, as taken directly from column C of the spreadsheet without formatting.
BitGroupRange\	The data from column C from the last ***#BitGroupDefinition*** line.
AddressH\	The same numerical value as *Address*\ but presented as hexadecimal number padded to the number of digits derived from the global AddressBitWidth (cell A32), with a leading "0X".
AddressH_NX\	Same as *AddressH*\ but without the leading "0X"
OffsetAddr\	The value of the register address from the last ***#RegisterDefinition*** line, plus the value of global cell InsertedAddressOffsetStr (A35), as a simple number.
OffsetAddrH\	Same as *OffsetAddr*\, padded to the size of the global OffsetAddrBitWidth, as a hex string with leading "0X"
OffsetAddrH_NX\	Same as *OffsetAddrH*\, but no leading "0X".
AltAddressB\	Same as *OffsetAddr*\ but padded to the number of bits digits as specified in the *alternate* address bit size specified in global cell A32, returning a binary string.
RegisterVHDLtype\	The data from column D from the last ***#RegisterDefinition*** line.
BitGroupVHDLtype\	The data from column D from the last ***#BitGroupDefinition*** line.
RegisterReference\	If column D is blank, same as *CurrentRegister*\, but if column D is not blank the value of column D.
GeneratedVHDLType\	If column D is blank, the value of global cell A39, but if column D is not blank the value of column D.
RegisterMode\	The data from column E from the last ***#RegisterDefinition*** line.
BitGroupModeStr\	The data from column E from the last ***#BitGroupDefinition*** line. This generates a multicharacter string as defined in Escape code *BitGroupModeStr*\ on page 27. If you want the data from column E as-is, use *E*\.
RegisterFunction\	The data from column F from the last ***#RegisterDefinition*** line.
BitGroupFunction\	The data from column F from the last ***#BitGroupDefinition*** line.
VHDLRegInitVal\ *VHDLRegInitVal:X*\	The value from column G from the last ***#RegisterDefinition*** line if that value is non-null. Otherwise, the value zero presented in binary to the global register bit width defined in cell A32. *VHDLRegInitVal:X*\ is the same, except the value in either case is a hex number.
RegReadMask\	A hexadecimal value padded to the number of bits defined in global cell A32, showing which bits of the register are defined as readable by having a *bitgroup* defined in the register whose directionality is readable.
RegWriteMask\	A hexadecimal value padded to the number of bits defined in global cell A32, showing which bits of the register are defined as writable by having a *bitgroup* defined in the register whose directionality is writable.
ConvertedBitGroupStr\	A string value of the form (n) or (x downto y) derived from the bit range of a bitgroup.
VHDLsubtype\	The value of column H of the row being processed. The column H data is expected to begin with a '.' character for concatenation with other strings to form something of the form xxxx.yyyy. If column H is not null and does not begin with '.', one is added at the beginning.
(BitGroupVHDLtype\ or *(RegisterVHDLtype*\	Yes, the "(" is part of the escape code. Uses the data from column D. If column D is blank, returns ".". If column D is non-blank and begins with "(", uses column D as-is. If column D is non-blank and does not begin with "(", returns "." Followed by the value in column D. Intended for forming array and/or record references in firmware. As the names imply, one is for use with bitgroups, the other with registers.

Escape Codes related to masking and shifting	
NUMBITSX\	This returns a two-character hexadecimal value without any leading 0X or &h that is equal to the number of bits wide the item being processed is. For a register the answer is defined by the global cell A29. For a bit group, it is the number of bits in the group.
STARTPOS\	For a register, 0. For a bit group, the position of the least significant bit of the group.
BITMASK\	For a register, the binary read or write mask calculated for the register. For a bit group, the binary read or write mask for the bit group.
HEXMASK\	The mask value that would be calculated for *BITMASK*\ but expressed as a hexadecimal value.
INVHEXMASK\	The one's complement of the value that *HEXMASK*\ would produce, also expressed in hexadecimal.
BG_shiftval\	The position of the bit, or the position of the rightmost bit within a bit group.
BG_binmask\	The binary bitmask that has a '1' in every position the bit group occupies with all other positions being '0'.
BITSHIFT\	For a register, always the string "0". For a bit group, same as *STARTPOS*\
XBITSHIFT\	The value of *BITSHIFT*\ expressed as a hex number. Generates nothing for a register.
Escape Codes related to formatting needs	
,\	Returns a comma unless the command being processed is ***#END_OF_DATA***, in which case a null string is returned. Intended for use in C templates that write out C structs.
,T\	Returns a comma unless the command being processed is ***#END_OF_DATA***, in which case a tab character is returned. Intended for use in C templates that write out C structs.
;\ or *;T*\	Identical to *,*\ and *,T*\, respectively, except the character returned is a semicolon, not a comma.
Escape Codes related to control system database generation Columns K through O of a row have special-purpose escape codes that are specifically designed to ease the generation of EPICS control system databases. The following rows describe those EPICS-specific escape codes. If you're not intimately familiar with EPICS, these escape codes will not make much sense to you.	
LABEL,VAL\, *INDEX,LABEL,VAL*\, *L,V*\ or *I,L,V*\	For bit groups defined as EPICS type binary (b) or multibit (m), takes the string in the **EPICS <idx>/label/value** column (column K) and parses that string as one of two forms. • <val>:<prompt>;<val>:<prompt>; for BI or BO process variables. <val> must be exactly 0 or 1; any other value is an error. The colons and semicolons are mandatory. • An unspecified number up to 16 of <idx>:<prompt>:<val>; triplets for MBBI or MBBO. Colons and semicolons are mandatory. The <idx> must be between 0 and 15. As the string is parsed the requisite number of EPICS field definitions for both label/prompt strings and value associated therewith are generated.
ZNAM\ *ONAM*\	When the user specifies a *Label,Val*\ for a single-bit database object (bi or bo), *ZNAM*\ and *ONAM*\ return the zero-value and one-value prompts extracted from the *Label,Val*\ decoding, respectively. Rarely used other than for making comments.
CONVERSIONS\	Using the same <xxx>:<yyy>; notation as used for *LABEL,VAL*\, this allows the user to insert any arbitrary number of field definitions into the generated PV that will read as *field(<xxx>,"<yyy>")*. The extractor reads the string from column L of the spreadsheet, different than that processed by the *LABEL,VAL*\ escape sequence. Double quotes around the <yyy> portion are automatically generated by the code and should not be entered into the spreadsheet. It is expected this would be used for EPICS value conversions or alarm limits. Example: the input string "F1:1;EPICS_THING:MY_DRIVER;ESLO:36.782;" will generate the following lines in the PV record related to the line of the spreadsheet being processed. *field(F1,"1")* *field(EPICS_THING,"MY_DRIVER")* *field(ESLO, "36.782")*
PV_NAME\	If the user has entered something in column M that string is used. If column M is blank, then the item name from column B is used. Useful if the firmware register name differs from the control system name for the same thing.
EPICS_ScanTime\ *EPICS_DTYP*\	Using the same <xxx>:<yyy>; notation as used for *LABEL,VAL*\, column N may define *override* values for the scan time or data type (DTYP) values (e.g., SCAN:1 second;DTYP:MyDataType;). When the line is read the override values are parsed out. These escape codes then will resolve to the override value if one was supplied, otherwise they return the default values specified in global cells A38 through A40.

USERCODE\	A more generic user input cell that is almost literally put into the generated output file as-is. The string as entered by the user is printed to the output file with the following substitutions: • the string "\t" is converted to a hard tab character • the string "\r" is converted to the carriage return character (ASCII 13) • the string "\n" is converted to a CR-LF (ASCII 13 followed by ASCII 10).
Generic Identifier\	A formatted string based upon the EPICS type of a line, of the form *Register : <registername> Address : <register address>* for a ***#RegisterDefinition*** line; for a ***#BitGroupDefinition line*** of the form *Bit: <name> position <n> of Register <registername> at address <registeraddress>* or *Bitgroup : <name> spanning range <range> of Register <registername> at address <registeraddress>* Intended for making formatted comments in output files.
Short Identifier\	A formatted string of the form *Reg:* or *Bit:* or *BGrp:* based upon the EPICS type of the line. Intended for making formatted comments in output files.
EPICS_BITMASK\	A special-purpose escape code for use in the AO.template file that translates to *HEXMASK*\ for objects of type "PW", but to *INVHEXMASK*\ for anything that's not "PW".
EPICS_TYPE\	This returns a string based upon the EPICS Type data in column J plus the directionality of the object being processed to form the standard EPICS record type strings (e.g. "AI", "AO", "BI", "BO", etc.)
EPICS_ASYN_BITMASK\	Returns a two-digit hex value indicative of the index of the lowest bit position used by a bitgroup. Used in conjunction with *NUMBITSX*\ to form masking strings of the format required by the EPICS "asyn" driver software.
IMPUTED_CHANNEL\	A special-purpose escape code for use only with the ***#EPICS_FANOUT*** keyword that allows the generation of PVs with channel numbers derived from the channel list in Column D.
Miscellaneous EPICS escape codes for odd formatting requirements These escape codes were developed to support the EPICS implementation of Argonne's "SBX" project (EPICS for a Raspberry Pi) and are unlikely to be useful to any other project. They are added for completeness of documentation only.	
SBX_BITMASK\	The string found in column R of the current row if cell is not blank, otherwise returns what the *BITMASK*\ escape code would return.
SBX_BITSHIFT\	The string found in column S of the current row if cell is not blank, otherwise returns what the *BITSHIFT*\ escape code would return.
SBX_XFR_MAILBOX\	Value of column D if not null, otherwise the string "0".
Tx\	If the current command is ***#EPICS_CALCOUT***, ***#RegisterDefinition*** or ***#BitGroupDefinition***, the value of column T. Deprecated.

Escape Codes related to the** #EntityDeclarationStart **and** #EntityDeclarationEnd **commands	
Entity_Declaration_Flag\	Used for *section headers*. Returns "Y" if the current line is between an **#EntityDeclarationStart** and an **#EntityDeclarationEnd**, "N" otherwise.
EntityDeclarationSkipCount\	Returns the current count of registers skipped within the #EntityDeclaration block.
EntityRegisterBaseName\	The value of column B from the **#EntityDeclarationStart** command.
EntityPortDirection\	The value of column C from the **#EntityDeclarationStart** command.
EntityVHDLType\	The value of column D from the **#EntityDeclarationStart** command.
EntityDescription\	The value of column F from the **#EntityDeclarationStart** command.
Escape Codes related to the** #ArrayExtractStart **and** #ArrayExtractEnd **commands	
ArrayFlagSet\	Used for *section headers*. Returns "Y" if the current line is between an **#ArrayExtractStart** and an **#ArrayExtractEnd**, "N" otherwise.
ArrayRegBaseName\	The value of column B from the **#ArrayExtractStart** command.
ArraySkipCount\	Returns the current downward count of registers within the ArrayExtract block. The count is initialized from column C of the ***#ArrayExtractStart*** line and counts down by one with each register processed. If the count drops below zero an error occurs.
ArraySkipCount-\	The value of *ArraySkipCount*\, minus one.
ArraySkipCountMax\	The original initialization value read from the ***#ArrayExtractStart*** line.
Escape Codes related to record, field, or signal definition commands	
SignalVHDLType\	The value of column D as read from the last ***#SignalDefinition*** line.
FieldInRecordVHDLtype\	Column D data as read from the last #FieldDefinition line.

TABLE 3 - SUMMARY LIST OF ESCAPE CODES USED WITHIN TEMPLATE FILES

Appendix 3: Syntax and functions of Assembly Files

Assembly files are text files with a file extension of **.assy** that are expected to be found in the same folder as the spreadsheet unless global cell A17 provides an alternate path. When the program enters the **Assembly** phase, all output files from processing spreadsheet data have been generated and closed. Thus the **Assembly** process is limited solely to manipulations of existing files.

Assembly File syntax

Assembly files may contain plaintext, comments (lines beginning with \\) and *Assembly directives.* The flow of an assembly file is that it consists of an arbitrary number of operations beginning with opening an *output* file, then writing things to that file, then closing it. Once a file has been opened, all plaintext within the assembly file is written to the output file, so plaintext may be used to insert starting and/or ending text in a similar way to template files used in the *Template lead-in* and *Template lead-out* phases of operation.

Assembly *directives* are not copied to the output but tell the assembly process to perform an action. By intermingling plaintext and assembly directives, the selected output file may be manufactured from an arbitrary number of spreadsheet outputs in whatever format is desired.

Assembly directive	***How it works***
\@OpenOutput@\ <filename>	Opens the specified file for writing. If the file already exists a warning is issued. If some other file is already open, the previous output file is closed before opening the new output file.
\@OpenOutputAppend@\ <filename>	Opens the specified file for append. If some other file is already open, the previous output file is closed before opening the new output file.
\@OpenInput@\ <filename>	Opens the specified file for reading. If some other file is already open, the previous input file is closed before opening the new input file.
\@SetInputPath@\ <string>	Changes the default path where input files are located. The default at start is "where the spreadsheet is".
\@SetOutputPath@\ <string>	Changes the default path where output files are located. The default at start is "where the spreadsheet is".
\@CopyInputToOutput@\	Copies previously opened input file to previously opened output file. During copying, special ***Break Sequences*** are recognized in the input file that suspend the copy process and return to processing the assembly file. See below.
\@CopyFileToOutput@\ <filename>	Opens the specified file and copies it verbatim to the current output file. This command may ***only*** be used following a \@CopyInputToOutput@\ to insert a file into the output stream after finding a ***Break Sequence*** of "SS_Insert" in the input file.
\@CopyFileToOutPath@\ <filename>	Opens specified file, copies it verbatim to the current output path. Intended to copy an assembled output file saved in the spreadsheet folder tree to some other place such as the tree of folders in a related project.
\@ResumeInputFile@\	Resumes copying of an opened input file to an opened output file in response to \@CopyInputToOutput@\ finding a ***Break Sequence*** of "SS_Insert" in the input file.
\@CloseOutput@\	Closes the current output file. The assembly process can have only ***one*** file open as output at any given time. Provided for completeness, not often necessary.
\@RenameFile@\ <old name> <new name>	Moves a file from one location on the disk to another.

TABLE 4 - LIST OF ASSEMBLY DIRECTIVES

File name specifications in assembly directives

File names in *assembly directives* may be specified using *relative* or *absolute* notation. *Absolute* file names are those containing a colon (":") in the file name string and are used as-is. *Relative* file names do not contain a colon and are treated as paths relative to the folder where the spreadsheet is.

Break Sequences in input files

When using the \@CopyInputToOutput@\ assembly directive, the assembly process is sensitive to a short list of special text strings called ***Break Sequences.*** These strings are selected to be bits of text that would be

interpreted as *comments* in various source languages so that they would have no effect on other code but be usable by the assembly process to insert spreadsheet-generated files at specific parts of another file.

Break sequences ***must*** be placed starting at column 1 of the input line, and matching is case-sensitive.

Break Sequence	***Response of* \@CopyInputToOutput@\ or \@ResumeInputFile@\ *thereto***
//SS_Stop, ##SS_Stop or --SS_Stop	Stops copying and closes the input file.
//SS_Pause, ##SS_Pause or --SS_Pause	Continues reading the input file but does not copy to the output file until SS_Resume, SS_Stop or EOF is seen. Used in conjunction with SS_Resume to block copying of a section of the input file.
//SS_Resume, ##SS_Resume or --SS_Resume	Continues reading the input file and resumes copying to the output file.
//SS_Insert, ##SS_Insert or --SS_Insert	Suspends the copy operation and returns control to the *assembly file*, allowing insertion of plaintext or other files using assembly directives. The copy operation is resumed by the directive \@ResumeInputFile@\.

TABLE 5 - RECOGNIZED BREAK SEQUENCES

For example, assume that the spreadsheet and its template files have been set up to generate an output file that is a complete VHDL subdesign with library use cases, generated packaged file, entity declaration, ports, and register logic. In a hierarchical design of this type there would be a parent that would require multiple sections of source code related to the register subdesign:

- Reference to any *package file* defining custom data types used by the register subdesign and/or a *component definition* for the register subdesign.
- A list of *signal definitions* for all the signals that will connect to the ports of the register subdesign, including any signals for whole registers, groups of registers or signals representative of the bitgroups within the registers
- The *instantiation* of the register subdesign itself
- Any *asynchronous assignments* that break out bitgroups of registers to separate signal names either for collection of things to be read or for assigning parts of registers for connection to ports elsewhere within the design

All of the above are easily generated by template files of the spreadsheet and would be available as individual files in the output folder when the Program Phase reaches *Assembly.* One may then write an assembly file of the form shown below. Note comments and plaintext; the comments will not be written, the plaintext will; the plaintext will make nice VHDL comments showing what was generated by the spreadsheet in the final output.

```
\@OpenOutput@\ merged_VHDL.vhd
\@OpenInput@\ C:\VHDLproj\top.vhd
\@CopyInputToOutput@\                    \\copies until --SS_Insert is seen or EOF
-------------------------------------------------
-- Library use cases generated by code-generating spreadsheet
\@CopyFileToOutput@\ SS_output\Usecases.VHDL_REG
-------------------------------------------------
\\extension VHDL_REG means register templates of "VHDL" language set.
\@ResumeInputFile@\
-------------------------------------------------
-- Signal definitions generated by code-generating spreadsheet
\@CopyFileToOutput@\ SS_output\Excelsig.VHDL_REG
-------------------------------------------------
\@ResumeInputFile@\
-------------------------------------------------
-- Instantiation of register subdesign generated by code-generating spreadsheet
\@CopyFileToOutput@\ SS_output\Instantiation.VHDL_REG
-------------------------------------------------
\@ResumeInputFile@\
-------------------------------------------------
-- Asynchronous assignments breaking out register bitgroups, generated by code-generating spreadsheet
\@CopyFileToOutput@\ SS_output\Async_assign.VHDL_BG          \\comes from bitgroup templates
-------------------------------------------------
\@ResumeInputFile@\
\@CloseOutput@\
\@RenameFile@\ SS_output C:\VHDLproj\merged_top.vhd
```

The above would open the user's VHDL file and insert four spreadsheet-generated files into an output file, then copy that generated file to the same folder as the user's VHDL file.

Appendix 4: Details of Section Header logic

Section Headers have previously been described for use within template files as simple yes/no comparison objects that may set or clear a flag whose value controls whether lines of a template file are processed. This Appendix lays out the rules of *section headers* in full detail.

When a template file is opened for processing the *SectionIsActive* flag is set to 1, meaning "copy enabled". Every line in the template file is read and if the line is neither a comment nor a *section header*, the line is processed for *escape code* substitutions and written to the output. Previously, in *Template File **Sections*** on *page 11*, the basic format of a *section header* was described, to wit:

\!TextToMatch!\ *EscapeCode*\

There are, however, additional varieties of *section header* provided that enhance the functionality.

Numeric comparison section headers

While *escape codes* always return *strings* for text processing purposes within template files, many of these are simply string representations of numerical values. Numeric *section headers* may be defined by replacing the "\!" and "!\" by "\#" and "#\", respectively. Section headers of this form take the string result of the escape code, convert it to a number, and then compare that number to the value of the section header. A "type of comparison" character is ***required*** between the "\#" and the match value, with the following effects:

- \#=53#\ *<escape code>*\ will compare the numeric value of the escape code to 53, and the *SectionIsActive* flag will be **set** if the returned value is **exactly** 53, **cleared** otherwise.
- \#>53#\ *<escape code>*\ will compare the numeric value of the escape code to 53, and the *SectionIsActive* flag will be **set** if the returned value is **greater than** 53, **cleared** otherwise.
- \#<53#\ *<escape code>*\ will compare the numeric value of the escape code to 53, and the *SectionIsActive* flag will be **set** if the returned value is **less than** 53, **cleared** otherwise.
- For all three cases above the user may specify a *second* escape code with *no* numeric value after the "=", "<", or ">". The logic will then compare the values of the two escape codes to each other (EC1=EC2, EC1>EC2 or EC1<EC2) and set/clear the *SectionIsActive* flag based upon the comparison of the two escape codes.

Additive/Subtractive section headers

Normal (non-numeric) *section headers* may optionally include a ***single*** function modulation character immediately after the "\!" and before the "text to match". The modulation characters are summarized below. Only one modulation character may be used, any characters after the modulation character are considered part of the "text to match". Only the first character after the "\!" can be considered the modulation character, the modulation characters seen later in the "text to match" string are not special.

Modulated section header format	***Effect upon SectionIsActive flag***
\!+TextToMatch!\ *escapecode*\	**Sets** *SectionIsActive* on match, does **not** change state of *SectionIsActive* on no match.
\!-TextToMatch!\ *escapecode*\	**Clears** *SectionIsActive* on match, does **not** change state of *SectionIsActive* on no match.
\!*TextToMatch!\ *escapecode*\	**Clears** *SectionIsActive* on match, **Sets** *SectionIsActive* on no match. (invert of normal logic)
\!>TextToMatch!\ *escapecode*\	Terminates processing of template file immediately if match, does **not** change state of *SectionIsActive* on no match.

TABLE 6 – ADDITIVE/SUBTRACTIVE SECTION HEADERS

Special function Section Headers

A small number of specific text strings are processed as special function section headers that do not require any escape code. These strings are case sensitive and must be matched exactly to take effect.

Special section header format	***Effect upon SectionIsActive flag***
\!ALWAYS!\	**Sets** *SectionIsActive* flag.

\!END_SECTION!\	**Clears** *SectionIsActive* flag.
\!EXIT_FILE_IF_ACTIVE!\	If the *SectionIsActive* flag is set, terminates processing of template file immediately.

TABLE 7 - SPECIAL SECTION HEADERS

Implementing simple section header logic to group similar templates together.

Additive and subtractive *section headers* may be grouped on adjacent lines of the template file to implement limited AND/OR logic, and if combined with the special function section headers allow for multiple similar templates to be grouped within a single template file. For example, a structure such as

```
\!R!\ \*RegisterMode*\
\!+FR!\ \*RegisterMode*\
\!+RW!\ \*RegisterMode*\
        --this is a readable object
\!END_SECTION!\

\!W!\ \*RegisterMode*\
\!+FW!\ \*RegisterMode*\
\!+PW!\ \*RegisterMode*\
\!+RW!\ \*RegisterMode*\
        --this is a writable object
\!END_SECTION!\
```

Will have the effect of outputting "--this is a readable object" if the RegisterMode is R or FR, outputting "--this is a writable object" if the RegisterMode is W, FW or PW, and outputting ***both*** lines if the RegisterMode is RW.

Appendix 5: Details of specialized firmware language support

The text in this section presumes that the reader is completely fluent in VHDL. No attempt to explain how VHDL code works will be made. While the vast majority of firmware generation functions are handled by template files and the assembly process, there are additional functions that the program supports that advanced users may find valuable.

Escape code *BitGroupModeStr*\

As bitgroups within a register are processed the combination of register directionality, bit group directionality, whether or not the bitgroup is enclosed within a #EntityDeclaration, and the ability to declare VHDL types and sub-types results in a bewildering number of possibilities that far exceed the ability *section headers* to select the appropriate format. There are many possible forms of bitgroup assignment within firmware that result, but they can be categorized. To provide support for the use of *section headers* to allow all the formats a project might use, the special escape code *BitGroupModeStr*\ is provided that returns a string value indicative of the many permutations that may occur. This escape code returns a 5-to-7-character string such as "YRWRWYN" that would translate as "bitgroup inside a #EntityDeclaration, whose RegisterMode is RW, whose BitGroupMode is RW, where user has overridden the RegisterVHDLtype default but the BitgroupVHDLtype is not overridden".

Included with this document is an example template file showing how every value of *BitGroupModeStr*\ may be used with section headers to generate an appropriately formatted line for all permutations.

Why use #EntityDeclaration

Register logic can be implemented in many different ways and it is difficult to generalize the process in such a way that will handle all possible use cases. The VBA code uses a model based upon the templating process discussed previously that is intended to be flexible enough to handle most cases. In the most general sense, register logic in a VHDL design will be coded in one of two ways:

1. The register design is implemented as *processes* for register read and register write operations that are placed *in-line* with other code within a VHDL file provided by the user.

2. The register design is implemented as a *sub-design* of the VHDL file provided by the user, in which case the *Port Map* connects *whole registers* as ports between the provided file and the generated sub-design.

These two models have some common requirements:

A. A *signal list* for all the *whole registers* must be generated and inserted into the user's VHDL.
B. Reference to any *package file* generated by the spreadsheet must be added to the user's VHDL.
C. As only the *whole register* objects are presumed to be the signals or ports, appropriate assignment statements must be generated for all ***bit-groups*** to assign them to the ranges of bits within the whole registers than they represent.
    a. This format may vary widely depending upon whether the *whole registers* are stand-alone signals or parts of VHDL *records* or *arrays.*

If whole registers are parts of *records* or are arrayed in some form, and the design does not separate the individual signals in the sub-designs port map, then appropriate formatting is also required for the ports of any generated sub-design.

Details of #EntityDeclaration

The #EntityDeclarationStart and #EntityDeclarationEnd commands are specifically intended to provide a mechanism for the creation of specialized *ports* when the spreadsheet is generating firmware intended to be a *sub-design* of a larger construct. As an example, presume that the spreadsheet is being used to develop the register logic for a design that has 10 "channels", such as a module with 10 ADCs whose data is being read and processed by a single FPGA. It would be logical to then have 10 "channel control" registers at 10 different addresses, but they likely would be formatted identically.

Some designers may wish to treat these 10 registers as separate objects throughout the design but others may wish to lump them together using code such as

```
type Arr_9_0_slv_31_0 is array(9 downto 0) of std_logic_vector(31 downto 0);
signal ChanControl : Arr_9_0_slv_31_0;
```

In this case, then, the *port* to a register subdesign must be of type Arr_9_0_slv_31_0, and there should be no *ports* generated in the entity declaration for the register subdesign for the individual registers. To accommodate this the **#EntityDeclarationStart** command may precede the block of **#RegisterDefinition** lines for the group of registers to be arrayed. **#EntityDeclarationStart** causes the spreadsheet program to

a. Set an internal flag variable stating that an entity declaration grouping is in progress;
b. Set an internal "skip count" to zero, to provide a counter to the template files of how many registers have been skipped within the group;
c. Process the **Register** template files in ***only*** the "VHDL" language set of template files, to allow those template files to output any preamble or alternate format line required.

After this, when each **#RegisterDefinition** or **#BitGroupDefinition** line is processed, the template files will be able to use special *escape codes* with *section headers* to modulate or suppress the generation of output data for the spreadsheet lines contained within the block as needed. When the **#EntityDeclarationEnd** command is processed, the flag variable is cleared and processing resumes as normal.

Package File Generation

The spreadsheet provides the commands **#RecordDefinition**, **#FieldDefinition**, **#RecordDefinitionEnd** and **#ArrayTypedef** to allow the user to embed type definitions within the spreadsheet. These simple commands just take information from cells in the row and use separate template files to write them into a package file in correct syntax. There is little simplification or processing here; this is just for convenience purposes. The following lines

show an example of how these may be used. If desired these lines, plus the **#LibraryDefinition** and **#LibraryUseCase** lines, allow the user to embed the design of a package file into the spreadsheet.

	A	B	C	D	E	F
121	**Fields used to define libraries ===>**	***Librarry/UseCase***				
122	#LibraryDefinition	ieee				
123	#LibraryUseCase	std_logic_1164				
124	#LibraryUseCase	std_logic_unsigned				
125	#LibraryUseCase	numeric_std				
126	#LibraryUseCase	math_real				
127	#LibraryDefinition	work				
128	#LibraryUseCase	function_lib				
129	#LibraryUseCase	ip_lib				
130	#LibraryUseCase	type_lib				
131	#LibraryUseCase	local_ip_lib				
132	#LibraryDefinition	unisim				
133	#LibraryUseCase	vcomponents				
134	**DEFINITION OF RECORDS**					
135	#RecordDefinition	t_DGS_TX_MACH_CONTROL				
136	#FieldDefinition		ASYNC_CMD_DATA	t_GRETA_COMMAND_FRAME	Data to send in the five words of Frame #3	
137	#FieldDefinition		MSM_MON_FIFO_SELECT_REG	std_logic_vector(15 downto 0)	Control register for monitoring FIFO	
138	#FieldDefinition		IMPERATIVE_FLAG_REQ	std_logic	If set, timestamp is held at start value	
139	#FieldDefinition		STARTING_TIMESTAMP	std_logic_vector(63 downto 0)	User defined starting timestamp value	
140	#FieldDefinition		ASYNC_CMD_REQ_FLAG	std_logic	Set when user wants to send GRETINA async message	
141	#FieldDefinition		FRAME_16_REQ_FLAG	std_logic	Set when user wants to send message	
142	#FieldDefinition		FRAME_16_DATA	array_of_slv_15_0(5 downto 1)	Array of registers holding data to send	
143	#FieldDefinition		FRAME_17_REQ_FLAG	std_logic	Set when user wants to send message	
144	#FieldDefinition		FRAME_17_DATA	array_of_slv_15_0(5 downto 1)	Connects to status register	
145	#FieldDefinition		TRIG_MON_DET_DATA	std_logic_vector(15 downto 0)	should not be a register; connection from timing module; connect later	
146	#FieldDefinition		TRIG_MON_XTRA_DATA	std_logic_vector(15 downto 0)	should not be a register; connection from timing module; connect later	
147	#FieldDefinition		PROPAGATE_SYNC	std_logic	should not be a register; connection fromDS92LV18 RX machine; connect later	
148	#FieldDefinition		DGS_mstr_mach_START_FLAG	std_logic	should not be a register; connection fromDS92LV18 RX machine; connect later	
149	#RecordEndDefinition	t_DGS_TX_MACH_CONTROL				

FIGURE 14 - DEFINITION OF VHDL LIBRARY REFERENCES AND RECORDS WITHIN THE SPREADSHEET

Use of record or array types in #RegisterDefinition and #BitGroupDefinition

Whether or not the user defines the records or arrays within the spreadsheet, registers and bitgroups may use columns D and H to specify types and subtypes, and then use *escape codes* such as *RegisterVHDLtype*\, *BitgroupVHDLtype*\ and *VHDLsubtype*\ to build appropriate formatting. It is possible to generate three-level record references (e.g. <thing>.<thing>.<thing>) with sufficiently creative use of the escape codes shown above plus use of ***#EntityDeclaration.***

Use of #SignalDefinition

#SignalDefinition uses its own template file and is intended to enumerate statements of the form ***signal <signame> : <vhdl type>;*** into a separate output file. Such enumerations of signals would normally be done by templates in the **Register** or **Bitgroup** subfolders to enumerate register signals that connect to ports or to break out bitgroups in asynchronous assignments, but in very rare cases the user may wish to add specific other signals or constructs into the design. Most often this command is never used.

Appendix 6: Details of specialized software ("C") language support

#RegisterDefinition lines generally cause one line to be written to the generated output files for software. There are times in test stand code where an array within a generated data structure would be preferred, and for this the *#ArrayExtract* commands are provided.

- ***#ArrayExtractStart*** sets an internal flag and loads internal variables ArrayRegisterBaseName and ArraySkipCount from columns B and C, respectively. The ArraySkipCount is the number of ***#RegisterDefinitions*** that should not generate any individual line in the generated output file, and is also used in writing the size of the array.
    - Because the ArraySkipCount is the size of the array, do not specify any braces or the array size in the array name.
- ***#ArrayExtractEnd*** closes the sub-structure definition and resets the array extraction flag.

By using *escape codes* that look at the state of the Array flag such as *ArrayFlagSet*\ or the skip count or by using *section headers* that key based upon column A being the specific commands #ArrayExtractStart or #ArrayExtractEnd, the user may implement whatever opening text or closing text is desired.

Appendix 7: Details of specialized control system (EPICS) language support

EPICS is a complex software program whose description is beyond the scope of this document. In a nutshell, EPICS provides a *distributed database* of *process variables* shared between an arbitrary number of computers on a network. Each computer (referred to in EPICS as an Input-Output Controller, or "IOC") has driver software and device support software that map reads and writes of hardware to the *process variables* (often called PVs). EPICS PVs come in a variety of types ('analog', 'bit', 'multibit') and have unitary directionality (either 'in' or 'out'), so process variable records are named 'ai', 'ao', 'bi', 'bo', 'mbbi' and 'mbbo'. An add-on to base EPICS called 'asyn' creates 'longword' PVs that are used for reading/writing integer hardware values to differentiate them from 'ai' and 'ao' that can be floating point; these are records of type 'longin' and 'longout'.

In the spreadsheet column I of every row may be filled with the letters A, B, M, L or X to indicate to the software that the PV to generate for this row should be 'ai'/'ao', 'bi'/'bo', 'mbbi'/'mbbo', 'longin'/'longout' or "don't make a PV". The escape codes for EPICS perform the translation to the record names and the templating logic allows generation of two PVs (one in the 'read' direction and one in the 'write' direction) for objects with a Register/Field Mode (column E) of "RW". Similarly section header logic within the template files can use the *EPICS_type*\ escape code to change formats based upon the specific format required for each PV record type.

EPICS Support related to columns K-O of each row.

With EPICS each PV is defined as a "record" that has "fields". For the 'bit' and 'multibit' types of PVs one must supply "fields" defining the prompts and values to write for each index available. 'bit' objects have only indices 0 and 1 whereas 'multibit' have indices 0 to 15. The 'analog' PVs, being floating point objects, have multiple "fields" defining the conversion that might occur (scaling, offset, addition of "units" for display). Sometimes the system requires that the name of the PV be different than the name of the object in firmware. All PVs have a "field" that specifies the name of the driver function they are associated with, and read-direction PVs must have a "field" specifying the rate at which they are scanned. Finally, different PVs may have additional "fields" not related to the hardware/firmware value but to the processing order within EPICS, so there must be a way to specify this. The data in columns K-O plus the different EPICS-related escape codes handle all these cases.

Column K : index/label/value

The data in this column consists of strings formatted either as doublets (for bitgroups with EPICS type 'B') or triplets (for bitgroups with EPICS type 'M'). The doublet form is ***0:label;1:label*** that defines the GUI label to provide in the definition of the PV for binary states 0 and 1. The triplet form is

idx:label:value;idx:label:value;…idx:label:value; for up to 16 different possibilities. The ***idx*** is the ordinal value (0-15) specifying the order in which the labels will be shown to the user in the GUI. The ***label*** in each triplet is the text of each label, and the ***value*** is the numerical value that the PV will take on for each index. The *I,L,V*\ escape code parses the string

```
0:zero:0;1:one:1;2:two:2;3:three:3;4:four:4;5:nine:9;6:fifteen:15
```

into the output

```
field(ZRVL,"0")
field(ONST,"one")
field(ONVL,"1")
field(TWST,"two")
field(TWVL,"2")
field(THST,"three")
field(THVL,"3")
field(FRST,"four")
field(FRVL,"4")
field(FVST,"nine")
field(FVVL,"9")
field(SXST,"fifteen")
field(SXVL,"15")
```

Column L : Conversions and units

The data in this column is processed by the *conversions*\ escape code. The user may enter text doublets (e.g. <string>:<string>) separated by semicolons and these will be parsed into EPICS process variable fields. The first string in a doublet is the name of the field, the 2nd string in the doublet is the value of the field. This is most commonly used with 'ai' and 'ao' process variables. As an example, entering the string

```
ESLO:0.010000;LINR:SLOPE;EGU:us;PREC:2;
```

Will generate the output

```
field(ESLO,"0.010000")
field(LINR,"SLOPE")
field(EGU,"us")
field(PREC,"2")
```

Column M : name override

Normally the escape code *PV_Name*\ will just return the name of the object from column B, but this escape code has a tiny bit of logic added to it so that if column M has data, then the escape code returns what is in column M instead. This allows template files to have two names for the name thing, should that be necessary.

Column N : DTYP/SCAN override

The escape codes *EPICS_DTYP*\ and *EPICS_ScanTime*\ relate to specific fields in an EPICS PV definition. Every PV requires a DTYP, and every read-direction PV requires a SCAN. There are global cells in the spreadsheet (A39 & A40 for DTYP, A38 for SCAN) that provide *default* values. The user may enter string doublets formatted the same way as done for column L to perform a one-line *override* of the default values with whatever's in column L. Thus if the default DTYP is "asynUInt32Digital" and the default SCAN is "1 second", entering **DTYP:myaltdriver;** in column L will cause the escape code *EPICS_DTYP*\ will be replaced by **field(DTYP,"myaltdriver")** instead of the default **field(DTYP,"asynUInt32Digital")** for the PV associated with just that one line in the spreadsheet.

Column O : Manual code

Inevitably special cases arise that are not handled by the functions of the software. This row may be used to generate whatever additional text is desired in the output file through the escape code *UserCode*\. The string entered in column O may contain the special character pairs "\n", "\r", and "\t" and these will be converted to line feed, carriage return and tab. Thus, I one enters the text

```
\tfield(SPECIAL,"17")\n\tfield(XTRA, "Hi mom")
```

Into column O, the resulting output is

```
field(SPECIAL,"17")
field(XTRA, "Hi mom")
```

Generation of large control systems using #EPICS_FANOUT

The spreadsheet defines all the registers and bitgroups of a given FPGA, but what if the end goal is to develop a control system that contains many copies of the same board with the same FPGA? For instance, the Digital Gammasphere system at Argonne National Lab consists of twelve VME crates, and each of these VME crates contains multiple digitizer modules. Additionally each digitizer module has 10 channels with identical control register structure for each channel (although all at different address offsets within the module). The users don't want to write lengthy scripts to set all channels of all modules in all crates to a given value or have to do that manually by clicking innumerable screens. EPICS has a support structure for this called ***fan-out PVs.***

Using the fan-out PVs, one may declare a "global" process variable that is hosted on the user's computer and then when the user sets that "global" PV to a value the value of the "global" PV connects through the fan-out PVs to all the other computers in the control system, but each of these computers then has to implement additional PVs to take the fan-out value and fan it out further to the channels of the boards within each crate.

System Definition spreadsheet cells

The spreadsheet implements a set of global cells (K1 through K8), of which cells K1, K2, K3, K5, K6 and K8 are specifically related to generating a system of multiple instances of process variables. For Digital Gammasphere the following setup is used.

K	L
DGS_systemdef.txt	System definition file name
GLBL:DIG:	fanout leader
JustGlobals.db	name of system global file
	assembly file error check level (default: blank, othwerwise a number)
	text to put in front of each crate name when forming file names
_dgs	text to put behind each crate name when forming file names
0	String replaces *UsrC*\ in templates.
	Name of special template file to make IOC boot scripts

FIGURE 15 - SYSTEM DEFINITION GLOBAL CELLS AS USED IN DIGITAL GAMMASPHERE

System Definition File

Cell K1 provides the path to a "system definition file" that is used to tell the spreadsheet what the architecture of the system is. This is a simple file that contains one line for each "crate" or "entity" within the system, and the format is ***<computername>:<board>;…<board>;***, as shown below. Gammasphere has twelve crates/computers named VME01 through VME12, and each crate has two or four digitizer boards with unique names for each.

```
VME01:MDIG1;SDIG1;MDIG2;SDIG2;
VME02:MDIG1;SDIG1;MDIG2;SDIG2;
VME03:MDIG1;SDIG1;MDIG2;SDIG2;
VME04:MDIG1;SDIG1;MDIG2;SDIG2;
VME05:MDIG1;SDIG1;MDIG2;SDIG2;
VME06:MDIG1;SDIG1;
VME07:MDIG1;SDIG1;MDIG2;SDIG2;
VME08:MDIG1;SDIG1;MDIG2;SDIG2;
VME09:MDIG1;SDIG1;MDIG2;SDIG2;
VME10:MDIG1;SDIG1;
VME11:MDIG1;SDIG1;MDIG2;SDIG2;
VME12:MDIG1;SDIG1;MDIG2;SDIG2;
```

This information may be used to differentiate PV names for each board in each crate. If we have a PV defined by the spreadsheet as **channel_control0**, then by prepending the computer and board name we may make unique PV

names for every channel 0 in every board in every crate, such as **VME01:MDIG2:channel_control0** or **VME10:SDIG1:channel_control0**.

Building fanout trees

Cell K2 defines the text that is prepended to PV names for the computer at the top of the fanout hierarchy and cell K3 defines the file name that will contain these top-level "global" process variables. As the spreadsheet processes, special lines with the command ***#EPICS_FANOUT*** are added to tell the software to build fanout trees for PVs of interest. For example, in the digitizer modules of Gammasphere there is a standard bit-group definition for a bit called ***master_fifo_reset***. There is nothing special about this, it is defined as a bit within a register like any other control bit:

B	C	D	E	F	G	H	I	J	K
Register/Field Name	***Address***	***VHDLType***	***Register/Field Mode***	***Description***	***VHDLRegisterIni***	***Fan***	***VH***	***EPICS***	***EPICS <idx>/label***
programming_done	0x0004		RW	Contains FIFO status and control				L	
unused	31..28		R	reserved				x	
master_fifo_reset	27		W	When asserted holds fifo reset.				B	0:run;1:reset

FIGURE 16 - DEFINITION OF MASTER_FIFO_RESET FOR GAMMASPHERE DIGITIZER

However, after the ***#EndRegisterDefinition*** that closes this register's design, we now add a new line with the command ***#EPICS_FANOUT***, like this:

A	B	C	D	E	F	G	H	I	J	K	L	M
EPICS_FANOUT_FIELDS	*Col. B: phases 1 and 2A, and also phase 2B if col. M empty*			*Channel s*	Board override							Phase 2b leaf PV name override
#EPICS_FANOUT	master_fifo_reset			Creates fanout tree					B	0:run;1:reset		master_fifo_reset

FIGURE 17 - BOARD-LEVEL USAGE OF #EPICS_FANOUT

This line *copies* the EPICS type ("B") and the index/label doublet ("0:run;1:reset") and specifies the name of the fanout PV in ***both*** column B and column M. When the software sees this line it uses the data from the System Definition File to write a series of PVs into the file whose name is specified in cell K3 (in this example, file *JustGlobals.db*). This is what we get:

```
##-------------------------------------------------------------------
##  Initial PV for GUI to drive fanout tree (#EPICS_FANOUT, phase 0)
##-------------------------------------------------------------------

record(bo,"GLBL:DIG:master_fifo_reset")
{
field(DOL,"0")
field(DTYP,"Soft Channel")
#  Source string : 0:run;1:reset
        field(ZNAM,"run")
        field(ONAM,"reset")

field(DESC,"Number of words containe")
field(OUT,"GLBL:DIG:F00:master_fifo_reset PP NMS")
}
##-------------------------------------------------------------------
##  Fanout of master_fifo_reset across crates in system   (#EPICS_FANOUT, phase 1)
##-------------------------------------------------------------------

record(dfanout,"GLBL:DIG:F00:master_fifo_reset")
{
field(OUTA,"GLBL:DIG:F01:master_fifo_reset PP NMS ")
field(OUTB,"VME01:GLBL:master_fifo_reset PP NMS ")
}
record(dfanout,"GLBL:DIG:F01:master_fifo_reset")
{
field(OUTA,"GLBL:DIG:F02:master_fifo_reset PP NMS ")
field(OUTB,"VME02:GLBL:master_fifo_reset PP NMS ")
}

... et cetera, et cetera ...
```

```
record(dfanout,"GLBL:DIG:F02:master_fifo_reset")
record(dfanout,"GLBL:DIG:F11:master_fifo_reset")
{
field(OUTA,"VME12:GLBL:master_fifo_reset PP NMS ")
}
```

As is seen, in *JustGlobals.db* a root PV **GLBL:DIG:master_fifo_reset** (name from cell K2 plus name from row being processed) is made, and then 'dfanout' PVs are made to fan this root PV out to a PV-per-crate. As the comments indicate, these PVs come from phase 0 and phase 1 of processing the ***#EPICS_FANOUT*** command. But this has to propagate further. There must ***also*** be a database file *for each crate's computer* and that file must continue to make fanouts to access every board in each of those crates. Looking in the output directory one finds *twelve* other files named **VME01_dgs.db** through **VME12_dgs.db.** The "VME01" through "VME12" comes from the System Definition file, the "_dgs" comes from global cell K6. In each of these files we find something like this:

```
##--------------------------------------------------------------------
##  Fanout of master_fifo_reset(#EPICS_FANOUT, phase 2A)
##--------------------------------------------------------------------
record(dfanout,"VME01:GLBL:master_fifo_reset")
{
field(OUTA, "VME01:MDIG1:master_fifo_reset PP NMS")
field(OUTB, "VME01:SDIG1:master_fifo_reset PP NMS")
field(OUTC, "VME01:MDIG2:master_fifo_reset PP NMS")
field(OUTD, "VME01:SDIG2:master_fifo_reset PP NMS")
}
```

This, then, completes the fanout to each board, as **master_fifo_reset** exists only once in each board.

Fanouts past boards to channels of boards

In the Gammasphere system, as noted before, each digitizer board has 10 channels. Because of the way the system is wired, two 10-channel boards service five detectors because each detector has four different signals that are digitized (GeC, BGOs, GeS and BGOp). It doesn't matter what those signal names mean, but the point is that the software must not only be able to generate another level of fanout to *channels* but *also* be able to sub-divide these fanout trees so that there's a global PV to drive just the GeC channels of a board and another for the BGOs channels *of the same board.*

This is accomplished by adding new information to columns D and E in the ***#EPICS_FANOUT*** command line and leveraging the dual names in columns B and M to have one name for the fanout *tree* (column B) but a different name for the final leaf of the tree that must match the *firmware name* of the target (column M).

A	B	C	D	E	F	G	H	I	J	K	L	M	N
#EPICS_FANOUT	BGOs_led_threshold		3	0;1;2;3;4;	MDIG1;MDIG2;	Creates write fanout tree			A		EGU:adu;	led_threshold	DTYP:Raw Soft Channel;
#EPICS_FANOUT	GeC_led_threshold		3	5;6;7;8;9;	MDIG1;MDIG2;	Creates write fanout tree			A		EGU:adu;	led_threshold	DTYP:Raw Soft Channel;
#EPICS_FANOUT	BGOp_led_threshold		3	0;1;2;3;4;	SDIG1;SDIG2;	Creates write fanout tree			A		EGU:adu;	led_threshold	DTYP:Raw Soft Channel;
#EPICS_FANOUT	GeS_led_threshold		3	5;6;7;8;9;	SDIG1;SDIG2;	Creates write fanout tree			A		EGU:adu;	led_threshold	DTYP:Raw Soft Channel;

FIGURE 18 - MULTILEVEL FANOUT DEFINITION THAT HANDLES BOARDS AND CHANNELS THEREOF

When this is processed the ***#EPICS_FANOUT*** creates four top-level fanouts named BGOs_led_threshold, GeC_led_threshold, BGOp_led_threshold and GeS_led_threshold that act as in the previous example. However, in the *crate level* fanout databases the information in column E is used to *sub select* only *some* of the objects within the System Definition file for the Phase 2A fanout, and then the column D data is used in a Phase 2B fanout to take the global and map it to only the enumerated list of channels for each board specified in column E. The result looks like this:

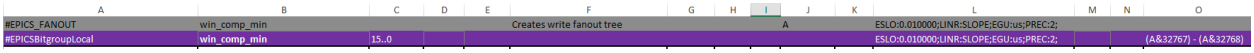
```
##--------------------------------------------------------------------
##  Fanout of BGOs_led_threshold(#EPICS_FANOUT, phase 2A)
##--------------------------------------------------------------------
record(dfanout,"VME01:GLBL:BGOs_led_threshold")
{
field(OUTA, "VME01:MDIG1:BGOs_led_threshold PP NMS")
field(OUTB, "VME01:MDIG2:BGOs_led_threshold PP NMS")
}
##--------------------------------------------------------------------
```

```
##  Channel sub-Fanout of BGOs_led_threshold for board MDIG1 (#EPICS_FANOUT, phase 2B)
##--------------------------------------------------------------------
record(dfanout,"VME01:MDIG1:BGOs_led_threshold")
{
field(OUTA, "VME01:MDIG1:led_threshold0 PP NMS")
field(OUTB, "VME01:MDIG1:led_threshold1 PP NMS")
field(OUTC, "VME01:MDIG1:led_threshold2 PP NMS")
field(OUTD, "VME01:MDIG1:led_threshold3 PP NMS")
field(OUTE, "VME01:MDIG1:led_threshold4 PP NMS")
}

##--------------------------------------------------------------------
##  Channel sub-Fanout of BGOs_led_threshold for board MDIG2 (#EPICS_FANOUT, phase 2B)
##--------------------------------------------------------------------
record(dfanout,"VME01:MDIG2:BGOs_led_threshold")
{
field(OUTA, "VME01:MDIG2:led_threshold0 PP NMS")
field(OUTB, "VME01:MDIG2:led_threshold1 PP NMS")
field(OUTC, "VME01:MDIG2:led_threshold2 PP NMS")
field(OUTD, "VME01:MDIG2:led_threshold3 PP NMS")
field(OUTE, "VME01:MDIG2:led_threshold4 PP NMS")
}
```

Use of #EPICSBitGroupLocal

While ***#EPICS_FANOUT*** generates control trees that allow a *write* to occur throughout a system, there are also instances where there is a need to introduce scaling to properly support floating-point values or to allow the end user to specify values in units other than the native units of the firmware. In Gammasphere, this is the case for the process variable **win_comp_min**. The user wants to think of this number in units of microseconds, but the firmware uses an integer number of 10ns clock ticks so scaling is required. The user also wants to think of this as a negative number, so sign bits come into play. An ***#EPICSBitGroupLocal*** line can provide the basic information as shown here.



A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
#EPICS_FANOUT	win_comp_min				Creates write fanout tree			A			ESLO:0.010000;LINR:SLOPE;EGU:us;PREC:2;			
#EPICSBitgroupLocal	win_comp_min	15..0									ESLO:0.010000;LINR:SLOPE;EGU:us;PREC:2;			(A&32767) - (A&32768)

And then a carefully crafted template file may then build a triplet of PVs to perform the needed conversion (read the data from hardware with a “longin” PV, link that to a “calcout” PV to handle the sign issue, then pass the result to an “ai” PV that does the scaling for the user to see what they want to see). The spreadsheet entry just provides the minimal data needed, the trick is that the ***#EPICSBitGroupLocal*** has its own special template file so these things are not generated for all registers, only the ones where they are needed.

The software also allows for commands ***#EPICSBitGroupGlobal*** and ***#EPICS_CALCOUT*** that have yet other custom template files to allow for whatever corner cases any given system may evoke. In the end, it’s just a hook to allow for a special template file for special cases.

Appendix 8 : Generation of GUI elements or other arbitrary outputs

As should be clear at this point templating can develop just about any output format desired. EPICS has various display manager programs, one of which is EDM (Extensible Display Manager). If one looks at the source files (extension .edl) for an EDM screen, each control is simply a bunch of text as shown below.

```
# (Button)
object activeButtonClass
beginObjectProperties
major 4
minor 0
release 0
x 387
y 10
w 22
h 16
fgColor index 25
onColor index 35
offColor index 3
inconsistentColor index 14
topShadowColor index 0
botShadowColor index 14
controlPv "VME32:MTRG:IMP_SYNC"
indicatorPv "VME32:MTRG:IMP_SYNC"
onLabel "R"
offLabel "S"
3d
font "helvetica-medium-r-18.0"
objType "controls"
endObjectProperties
```

Most things are fixed and the only variables are the x/y position, the 'controlPV', the 'indicatorPV' and maybe the colors. If this text is put into a template file the 'controlPV' and 'indicatorPV' obviously may come from the spreadsheet, and if you want you could take any columns of the spreadsheet desired to encode the x/y position or the colors. Or just use *inrowx10*\ to space them out equally.

By the same token use of the *EPICS_type*\ in the section headers of the template allows the spreadsheet to automatically generate a simplistic screen with a GUI control for every line while the spreadsheet is also generating the EPICS databases. Aesthetics aside, this at least gets the display started.

Whether the display manager is XML or CSS or HTML, most of these are text formats, so don't limit your creativity to just code, the spreadsheet can start off the user interface too.

Appendix 9 : File Search Function

There is a separate VBA macro included in the spreadsheet named **XXX_hunt_old_files** that may be invoked by the user that may be used in conjunction with global cells A42, A43 and A44. When the **Extract_Registers** program is run, it looks at cell A42. If the first character of cell A42 is the letter V, then as all the lines are processed the value in column B is copied to column AA. If A42 is blank then the "PV_name" (same as you get from *PV_Name*\ is put in column AA. Objects with a type of 'X' don't copy to column AA. This forms a search list of names to look for in files.

Cells A43 and A44 allow the user to specify a path and file wildcard that defines a list of search files. Use Windows Explorer to find the folder of interest and copy/paste into A43 to define the absolute path. Invoking **XXX_hunt_old_files** will search every file defined by cells A43 & A44, for every entry in column AA, and will list all the matches found in columns AB and forward. In other words, it's like "grep", but hunting for object names within source.

There's also a hook in this code where if the string in cell A42 is longer than one character, the program will take each input file and convert line-feed characters to carriage return/line feed pairs, to allow the search function to work on files taken from a Linux system.

By setting cells A42, A43 and A44 as needed, this functionality may verify where names are referenced in firmware, software, database or other source. It can help track down errors.

Licensing : It's free, it's provided as-is, don't expect perfection.

All VBA source code has been tagged with the "unlicense", as follows:

```
'=====================================================================
'This is free and unencumbered software released into the public domain.
'
'Anyone is free to copy, modify, publish, use, compile, sell, or
'distribute this software, either in source code form or as a compiled
'binary, for any purpose, commercial or non-commercial, and by any
'means.
'
'In jurisdictions that recognize copyright laws, the author or authors
'of this software dedicate any and all copyright interest in the
'software to the public domain. We make this dedication for the benefit
'of the public at large and to the detriment of our heirs and
'successors. We intend this dedication to be an overt act of
'relinquishment in perpetuity of all present and future rights to this
'software under copyright law.
'
'THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
'EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
'MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
'IN NO EVENT SHALL THE AUTHORS BE LIABLE FOR ANY CLAIM, DAMAGES OR
'OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,
'ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
'OTHER DEALINGS IN THE SOFTWARE.
'
'For more information, please refer to <http://unlicense.org/>
'=====================================================================
```

The author has every expectation that anyone who uses this software will find reason to change it and users are encouraged to fix any bugs they find or make improvements however they wish. No protections have been placed on any of the VBA source code.