# Generation of C and VHDL from spreadsheet

Usage and coding notes, 20191023, J.T. Anderson

The ‘spreadsheet as sole source of register design’ concept has been extended in October of 2019 to include many more features than it used to have. The Visual Basic for Applications (VBA) code embedded within the Excel spreadsheet now writes significant amounts of VHDL for you based upon the generic registers design developed by Michael Oberling. This document is intended to describe the new features, explain the different files generated by the code when it runs and provide hints where possible about how to use it.

Contents

[Generation of C and VHDL from spreadsheet 1](#_Toc22813346)

[File Structures 1](#_Toc22813347)

[The Python Flow 1](#_Toc22813348)

[The VBA Flow 1](#_Toc22813349)

[Data Model 3](#_Toc22813350)

[Implications of the data model on test stand software, end-user software & firmware 3](#_Toc22813351)

[Spreadsheet Structural Rules 5](#_Toc22813352)

[Summary of column usage 6](#_Toc22813353)

[How to generate registers (the basics) 7](#_Toc22813354)

[Generated VHDL Code Examples 8](#_Toc22813355)

[More advanced register generation tricks 9](#_Toc22813356)

[Control String Details 10](#_Toc22813357)

[# LibraryDefinition, # LibraryUseCase, # LibraryEndDefinition 10](#_Toc22813358)

[#RecordDefinition, #FieldDefinition, #RecordEndDefinition 10](#_Toc22813359)

[#ArrayTypeDef 11](#_Toc22813360)

[#EntityDeclaration 11](#_Toc22813361)

## File Structures

The root source of register definition is intended to be the spreadsheet. The spreadsheet is processed not only by VBA code within it but also by external Python scripts. There are two different flows with different files generated in each.

### The Python Flow

The Python scripts translate the data of the spreadsheet into various files needed by the embedded VME processors of Digital Gammasphere, DFMA, Clover, X-Array, etc. as shown in Figure 1.



Figure - The spreadsheet-Python path for EPICS & VME

A VBA script within the spreadsheet called *Check\_Python* does a sanity check of the data within the spreadsheet and will flag formatting issues (commas in fields, incorrect blank fields, etc.) that are known to cause the Python scripts to fail or crash. A spreadsheet that passes *Check\_Python* with no errors usually is processed by the Python scripts without incident.

As of 20200511 the Python flow is required to generate *some* of the pieces necessary to build the software that runs in the VME processors of Digital Gammasphere, but not *all* of the code. The general idea behind the design of the VME processor software is that most of it, once written and debugged, doesn’t change. Thus a modular approach that rewrites only the sections associated with register I/O to the VME cards is the basis behind the Python scripts.

### The VBA Flow

The VBA Flow generates all necessary files to integrate a register description into FPGA source code and LabWindows usage of that FPGA code from the spreadsheet. On the LabWindows side, C source and header files are generated that form the necessary data structure and initialization of same to allow for register-level access of a board implementing the registers of the spreadsheet. On the FPGA side, multiple files are generated, and then some of those files are recombined to generate a complete and ready-to-compile VHDL design. From a template top-level VHDL file and the spreadsheet data the VBA code generates

* A full VHDL registers subdesign, ready to compile;
* A package file defining any and all data types (arrays, records, etc.) defined within the spreadsheet;
* A 2nd package file providing a component definition of the registers subdesign that exactly matches the ports of the subdesign;
* A modified top-level VHDL file that is the template with the generated register design fully instantiated and all ports connected to signals of the same name as the ports, plus definition of all the signals that are connected to the ports in the architecture header.

Thus, once the FPGA project has been set up to use the generated top-level file, the generated register subdesign, the generated type definition package and the generated component definition package, registers may be added, deleted or modified in any way desired in the spreadsheet ***and all the changes needed in the VHDL are automatically generated within seconds.***

This overall file generation flow is shown in Figure 2.



Figure - Files generated by the VBA flow.

As of 20200511, the VBA flow is being expanded to generate the PV database portion of the EPICS output, replacing one part of the Python flow. The reason behind this is that there are now three different forms of EPICS PV database that need be supported by the spreadsheet model :

* Generation of PVs for GRETA from spreadsheets defining the trigger and timing modules developed for that project;
* Generation of PVs for Raspberry-Pi based EPICS implementations such as the Slope Box Extension, the Collector Box and other Gammasphere improvements.
* Generation of PVs for Digital Gammasphere usage (potential decoupling from Python)

# Data Model

To operate and communicate with hardware, there is software that has to communicate with the module. That software will have some kind of data model by which it translates “objects” into the physical design of the module. By definition all FPGA-based designs at the physical level will collect together all control and monitoring data into *registers.* Such *register* objects are mostly read-write objects, although there are many cases of read-only registers (e.g. status bits, counters) and occasionally there are write-only registers (collections of reset bits or one-time enables to state machines). All the read-write registers (control objects) are usually required to be implemented such that whatever was last written to them can be read back at any later time. Registers may be singular data values or their width may be arbitrarily subdivided into *fields* of one or more bits. To keep it simple, fields are always assumed to be contiguous groups of bits.

Practically, within the FPGA, registers have to be implemented using a *shadow RAM* – that is, a memory buffer that holds a copy of all the values written in one place. This eliminates the need for vast numbers of multiplexers spread throughout the design physically routing the actual flip-flops updated by writing the register back to the data bus for the read-back requirement. Implementation of the shadow RAM can easily reduce FPGA logic/routing requirements by 15% or more and makes timing closure vastly simpler.

As soon as the firmware designer makes that obvious choice, all register objects will be the same width irrespective of the number of bits actually used. This means that all unused bits in a register can simply be masked off in the read-back from the shadow RAM, and of course unused bits means no logic generated anywhere else for that bit. The reductionist (engineer) data model is an arbitrary number of identically-sized physical objects (the full registers). To support this model the VBA code generates .c and .h files for use in engineer-developed test stands. Full-register EPICS PVs can optionally be generated.

There is also the user/programmer model from EPICS, in which registers are not thought of, but instead “process variables” or PVs. How such PVs map to physical registers is preferably hidden by driver software so that the user/programmer doesn’t know what fields are in which register. All is buried in the device driver. In a parallel bus system like VME that width is naturally derived from the width of the backplane data bus. In a serial control system like Ethernet, some convenient size that aligns nicely with the packet definition is used (bytes, words or longwords – 8, 16 or 32 bits).

An inevitable outcome of actual hardware design is that all control and all monitoring transactions must always operate on a whole register. Thus even though a programmer or an end user might think that a field within a register is somehow a distinct object such is never the case; a write of a “field” always means there is a read-modify-write happening somewhere deep within the driver level.

## Implications of the data model on test stand software, end-user software & firmware

Because fields are not separable physical transactions but always conventions overlaid on top of the actual register-at-a-time physical transactions, the various extractions from the spreadsheet immediately split into “field” extractions and “register” extractions. Examples of this are as follows.

* Test stand code wants a list of registers and their addresses but has only low to moderate interest in fields. The transactional portion of the test stand code only uses registers so the physical interface devolves to exactly two functions: WriteRegister(address, data) and ReadRegister(address, \*data).
* An EPICS database for the end user wants to know all about fields but has little interest in register addresses. Only the driver underlying the EPICS database is interested in registers, and the driver has to have a consistent method of correlating fields to registers both for writes and for reads.
* Inside the FPGA, the VHDL code typically implements registers as std\_logic\_vector objects. Most of the time the code will simply extract the equivalent of a field by a sub-vector expression (x downto y). But, VHDL does have the concept of *records* that are essentially identical to C *structs*. So if the VHDL code is *record-based* all the “field” concepts can be propagated downward and this would make the VHDL easier to read and maintain. However, since historically most VHDL is manually generated that won’t happen without a tool.
	+ With the use of this spreadsheet and the embedded VBA code therein, firmware developers can much more easily define individual registers as *records* so that in the VHDL usage of those registers elsewhere, the firmware designer may use a *name* as opposed to a bit index or range. This not only makes the code more readable but also ensures that it is not possible to accidentally use a bit or range of a register for multiple functions.
	+ The second way this spreadsheet system enables the use of records is by making it easy to define a record data type consisting of multiple values that would span multiple registers. The group of registers within the record get passed as a record, and again all field names propagate into the VHDL for easier to read code.

# Spreadsheet Structural Rules

 The spreadsheet contains many lines and for each line information from many columns is used to define the output. Each line is processed independently. The main rules are:

1. Because Python uses the tab-separated text version of the spreadsheet file and not the spreadsheet directly, no commas are allowed in any field. Because the Python code is not carefully written, no tab characters are allowed in any field.
2. The spreadsheet looks for the specific string ***%HDR*** in column A and will use the row after the row in which that string is found as the starting row. The spreadsheet looks for the specific string ***#END\_OF\_DATA*** in column A and will stop processing at that row.
	1. The spreadsheet displays the number of rows to process and the starting row in cells B2 and B3, respectively. *Do not enter data into these cells. Verify that the values are sensible before running the VBA macros.*
3. The data value stored in cell E2 of the spreadsheet is the base file name that will be used in making the VHDL register design, the component package file and the types package file.
	1. The data value in cell E3 is the base of the file name for all the test stand .c and .h generated files.
	2. Cell S2 contains the path from where the spreadsheet is located to where the template top-level VHDL file is to be found. By definition the generated top-level file with merged-in signal definitions and register subdesign instantiation is always made in the same folder as the spreadsheet itself and is always named ***generated\_top.vhd.***
4. Column A of each row is the main control over how the row is to be processed. If column A is blank then the row is processed as either a *whole-register* line (column A blank, columns B/C/D have data) or a *field* line (columns A/B/C/D all blank).
	1. If column A is not blank then the data in column A is compared against a set of *control string* values to see if the line should be processed in certain defined ways. The list of defined *control strings* is as follows:
* #LibraryDefinition
* #LibraryUseCase
* #LibraryEndDefinition
* #RecordDefinition
* #FieldDefinition
* #RecordEndDefinition
* #ArrayTypedef
* #EntityDeclaration
* #ArrayExtract
* #C\_Comment
* #VHDL\_Comment
	1. If the data in column A doesn’t match any of the predefined *control strings,* then the entire line is not processed and can be used as a human comment.
1. Registers are defined by an arbitrary number of *field* lines that describe bits or bit-groups within a register and *whole-register* lines that define the register itself.
	1. All the *field* lines have to be in a contiguous group of rows. Non-active human comment lines may be intermingled if useful.
	2. The *whole-register* line must be the row immediately following the last *field* line.
2. *Field* lines that define a group of bits within a register but such groups must always be
	1. *Formatted* as n..m where ‘n’ is the *larger* number (i.e. 17..5 is ok but 4..9 ***is not***)
	2. *Contiguous* with no skipped bits (i.e. 12..8 is ok but you can’t specify something like 12..10;8..3 or any other non-contiguous single group).
3. Due to the fact that the VBA code processes the spreadsheet linearly from the top with only one pass, any uses of the # LibraryDefinition, # LibraryUseCase or # LibraryEndDefinition control strings ***must be the first active lines in the spreadsheet before any other active lines.*** This is driven by the way VHDL code is structured; library definitions must be at the top.

## Summary of column usage

 It is probably far easier to just look at a few examples as provided below, but as a reference the following table summarizes which columns are important to the Python scripts and which are important to the VBA code. In general, columns B through R are the columns Python uses. The VBA code also uses many of those same columns but also uses columns S through Z for VHDL-specific operations.

|  |  |  |  |
| --- | --- | --- | --- |
| ***Column*** | ***Name*** | ***Python Usage*** | ***VBA Usage*** |
| A | Comment/Control | Flags lines as comments. | Various key phrases are used to create different outputs. |
| B | Register Address | Passes through to EPICS database. | Used in generation of VHDL. |
| C | Register Mode | Sets up IN vs. OUT in EPICS | Modifies syntax of VHDL. |
| D | Register Name | If non-blank causes generation of a “whole register” PV entry. | Used to generate LabWindows .c and .h file lines. Also part of VHDL written to generated subdesign. |
| E | Register Function | Used as comment field in PV database. | Used as comment field in VHDL and C. |
| F | Access Type | ***Uncertain, but think it matters. Believe Python wants values of R, A or F.*** | Not used by VBA. |
| G | Address – *is copy of Register Address for all fields of register* | Used in generation of C driver portion for VME processor. | Not used by VBA. |
| H | Bit Range | Used in generation of PV. | Used to define sizes of std\_logic\_vectors and assignment of record fields to bit ranges of registers. |
| I | Field Mode | *Thought to be used in PV generation, not certain.* | Used in assignment of record fields to bit ranges of registers. |
| J | EPICS type | Sets type of PV to generate. ‘X’ means “do not process this line”. | ‘X’ means “write comment, do not make assignment”. |
| K | EPICS units | Used to create scaling factors for numerical PVs. | Unused by VBA. |
| L | Database Group | Usage unknown. | Unused by VBA. |
| M | Initial Value | Usage unknown. | Unused by VBA.[[1]](#footnote-1) |
| N | Register Name (2) – understood to be copy of column D of whole-register line for all field lines of that register. | Apparently simplifies Python extraction of PVs for fields as opposed to extraction of PVs for registers. | Unused by VBA. |
| ***Column*** | ***Name*** | ***Python Usage*** | ***VBA Usage*** |
| O | Software field name | Becomes name of PV. | Used to generate variable names in VHDL for non-record field types. |
| P | Bitfield Sub-Descriptor | Used to create PV value lists for enumerated PVs | Unused by VBA. |
| Q | Human field name | Becomes the GUI label of the PV | Unused in VBA. |
| R | Field Description for database | Transfers as comment. | Transfers as comment. |
| S | VHDLArrayBaseName | Unused by Python. | Used in writing library statements and in writing of package file. |
| T | VHDLRecordBaseType | Unused by Python. | Used in most VHDL generation functions. |
| U | VHDLRecordFieldName | Unused by Python. | Used in most VHDL generation functions. |
| V | VHDLRecordFieldType | Unused by Python. | Used in most VHDL generation functions. |
| W | ForceWholeRegVHDL | Unused by Python. | Forces special handling of whole-register definition lines for registers that have no fields defined. |
| X | RegEntityDefVisible | Unused by Python. | Used to flag whether whole-register definition lines are processed into lines in the entity and component declarations, to allow use of records. |
| Y | FanInGroup | Unused by Python. | If filled in, forces registers into a particular read-back fan-in group; can provide more compact code than default automatic assignment by putting all registers associated with similar physical location in FPGA into same group. |
| Z | VHDLRegisterInitValue | Unused by Python. | Defines power-on value that register will have if filled in. |

# How to generate registers (the basics)

 A register is simply defined as a set of fields followed by a whole-register definition. Here’s a relatively straight-forward example.



Figure - A straight-forward register definition

The lines above define a register named TRIGGER\_PRESCALE\_C that will be implemented at address 0x388. The 32-bit register only uses bit 15 (the field TRIG\_C\_PRESCALE\_ENBL) and also bits 7:0 (field TRIG\_C\_PRESCALE\_FACTOR). The remaining bits (14:8 and 31:16) are enumerated as two more un-named fields. The ***Register Mode*** field in column C, next to the ***Register Address*** in column B, states that the register as a whole is a read-write object (the most typical kind of register). The four white rows define the fields, the yellow row is the whole-register object.

When the VBA reads the four *field* lines it looks at the ***Field Mode*** (column I) and the ***Epics*** Type (column J) to choose what is done. The coding for ***Epics Type***whether any VHDL is to be generated. A value of ‘X’ means “do nothing”. The other values have meaning for the Python code but are meaningless to the VBA. If the ***Epics Type*** is not ‘X’, then the ***Field Mode*** determines the type of VHDL that is written. A value of ‘R’ means this is a read-only field, a value of ‘W’ means this is a write-only field and a value of ‘RW’ means read-write. If the type of ‘W’ is chosen this implicitly means that the bit is a transient, or *pulsed control*, bit; it will take on the value that is written but self-clear within a very short time.

Similarly the Python code uses the ‘X’ rule to enable/disable generation of process variables (PVs) for each field row. The ***Field Mode*** defines whether an output PV with the name taken from the ***SoftwareFieldName*** column is generated, and also whether a separate input PV with the name of the field plus “\_RBV” (for ReadBackValue) is also generated. The ***EPICS Type*** column defines the type of PV that is made. When the whole-register line is processed, the Python code generates another PV or two (depends on read/write setting) with the name “reg\_XXXXX” where XXXXX is the value in the ***Register Name*** column. To a degree the Python code also tries to set up the GUI for the EPICS code. The ***HumanFieldName*** column data is stored in the generated PVs as the text/legend for the PV in whatever display manager is used. The ***Field Description*** value is stored in the PV comment field.

Prior to 20200630, the code was written to automatically generate a specific set of libraries. To increase flexibility in its usage, there is currently a feature that reads text files of information not found on the spreadsheets into the generated files (namely, libraries). A group of cells at the top of Column P have been designated to accept the file path of: a library text file, a set of initial ports, and any other information to be added on at the end of the registers file.

## Generated VHDL Code Examples

* In the ***converted\_top*** top-level VHDL, a signal line is generated for the whole register:

signal TRIGGER\_PRESCALE\_C : std\_logic\_vector(31 downto 0); -- Prescale control for algorithm 3

* Also in ***converted\_top***, a line is generated in the instantiation of the register subdesign:

TRIGGER\_PRESCALE\_C => TRIGGER\_PRESCALE\_C, -- Prescale control for algorithm 3

* In both the register design file and the component definition package file, a line is generated defining the port connection:

TRIGGER\_PRESCALE\_C : INOUT std\_logic\_vector(31 downto 0); -- Prescale control for algorithm 3

* Within the register design itself, lines are generated that not only define the register as a whole but also how all the fields break out:

(X"388", X"00000000", 94, X"000080FF", X"000080FF", 6 ), -- TRIGGER\_PRESCALE\_C

And

TRIGGER\_PRESCALE\_C(7 downto 0) <= reg\_out(a\_to\_i(R, X"388")) (7 downto 0); -TRIG\_C\_PRESCALE\_FA…

-- No Field Generated for (14..8) from line 787 of spreadsheet; EPICS type is X

TRIGGER\_PRESCALE\_C(15) <= reg\_out(a\_to\_i(R, X"388")) (15); -- TRIG\_C\_PRESCALE\_ENBL (Set to…

-- No Field Generated for (31..16) from line 789 of spreadsheet; EPICS type is X

A little explanation of the register as a whole VHDL line (colored in red above) is needed. The line is defining an array of values that are used to define a custom data structure in VHDL. The elements of the line are, in order,

* Address of the register
* Power-up initialization value of the register (hex)
* Simple index count of how many registers have been defined (first is 1, then 2, etc.)
* The mask value to use for read-back (hex; unimplemented bits are ANDed with 0s in the mask)
* The mask value to use for writing (hex; unimplemented bits are ANDed with 0s in the mask)
* The read-back fan-in group number (used to control generation of multiplexer logic)

If the whole-register line has a value specified for ***VHDLRegisterInitialValue*** (column Z), that value is put into the VHDL. The value in column Z is assumed to be hex. The user may enter the value in column Z as just the value string (e.g. 0D37F09A) or may put an X or 0X in front if they like. The code handles the prefix.

## More advanced register generation tricks

 The data type for a register in the entity and component declarations defaults to *std\_logic\_vector(31 downto 0)*. If you want something else, that can be overridden by putting the desired alternate text in column V, ***VHDLRecordFieldType.*** For instance, VHDL code may be made more readable through the use of records. A record data type can be defined that matches a particular register so that all the names of the fields can be used throughout the VHDL instead of hand-typing bit indices. In that case, fill column V with the name of the record data type and that’s what will be put in the entity and component declarations.

 Continuing that logic, for *fields* the spreadsheet allows the user to put data in column T (***VHDLRecordBaseType***) and column U (***VHDLRecordFieldName***). In the example above these columns were blank so the VBA code just extracted the bit indices when writing the VHDL. But if the user fills in these columns then the assignments to reg\_in and reg\_out are replaced by VHDL record syntax of

< ***VHDLRecordBaseType>.<VHDLRecordFieldName>***

An example of this is found in the GRETA trigger design where a few field definition lines of the spreadsheet look like this (some columns hidden for clarity)



Plus a whole register line of (again, columns hidden for clarity)



The VHDL these field lines generate looks like this:

REG\_1GBPS\_DATAPATH\_CONTROL.LOCAL\_LE <= reg\_out(a\_to\_i(R, X"038"))(4); -- Local loopback enable

REG\_1GBPS\_DATAPATH\_CONTROL.LINE\_LE <= reg\_out(a\_to\_i(R, X"038"))(5); -- Line loopback enable

REG\_1GBPS\_DATAPATH\_CONTROL.SYNC <= reg\_out(a\_to\_i(R, X"038"))(6); -- If set drives sync…

REG\_1GBPS\_DATAPATH\_CONTROL.TTC\_DOUT\_SEL <= reg\_out(a\_to\_i(R, X"038"))(7); -- Mux select…

With the entity declaration reading as

REG\_1GBPS\_DATAPATH\_CONTROL : INOUT t\_DS92LV\_PATH\_AND\_CHIP\_CTL; -- controls source of 1Gbps…

With, of course, matching lines in the package file for the component definition and in the converted-top signal definition area. Thus the use of the additional columns allows use of records throughout the VHDL design.

# Control String Details

 The various control strings are described in more detail here. For each control string or group of related control strings examples from a working spreadsheet are provided.

## # LibraryDefinition, # LibraryUseCase, # LibraryEndDefinition

## #RecordDefinition, #FieldDefinition, #RecordEndDefinition

These lines are usually colored light blue in the spreadsheet by convention and use only columns A, R, T, U and V.



Figure - Example of defining a VHDL record with un-needed columns hidden

Lines with these key values in the ***Comment*** column (column A) create lines of the form shown in the box below into the generated package file.

-- Record containing DS92LV18 controls

Type t\_DS92LV\_PATH\_AND\_CHIP\_CTL is record --

 DEN : std\_logic; -- Driver Enable to DS92LV18

 REN : std\_logic; -- Receiver Enable to DS92LV18

 RPWRDN : std\_logic; -- Active low power down for receiver half

 TPWRDN : std\_logic; -- Active low power down for xmit half

 LOCAL\_LE : std\_logic; -- Local loopback enable

 LINE\_LE : std\_logic; -- Line loopback enable

 SYNC : std\_logic; -- If set drives sync pattern x00FF

 TTC\_DOUT\_SEL : std\_logic; -- Mux select for 1Gbps data (0:DS92 1:FPGA)

 DATAPATH\_RESET : std\_logic; -- Active high reset

 DATAPATH\_MODE\_REQUESTED : std\_logic\_vector(2 downto 0); -- TX mode desired

 DATAPATH\_SWITCH\_FLAG : std\_logic; -- pulsed control to request mode switch

end record t\_DS92LV\_PATH\_AND\_CHIP\_CTL;

If there is any text in column R (Field Description For Database) that will be passed through as a comment. As is plainly obvious these lines are just read, slightly formatted and spit back into one file, so the only true value of lines of this type is that they provide reminder to the user how the records are formatted when writing register definitions. In that sense they do reduce human error.

## #ArrayTypeDef

 Lines of this type read the data in column E (Register Function), column H (Bit Range), column S (VHDLArrayBaseName) and column T (VHDLRecordBaseType) and output a line to the package file of the form

***Type <***VHDLArrayBaseName***> is Array(<***Bit Range***>) of <*** VHDLRecordBaseType***>;***

Again, this is not much different than just typing the line into the package file yourself, but it can be useful to keep track of type names when defining registers in the spreadsheet.

## #EntityDeclaration

 Lines of this type read the data in column E (Register Function), column T (VHDLRecordBaseType), column U (VHDLRecordFieldName) and column V (VHDLRecordFieldType) and output a line to the generated subdesign file and the generated component declaration file of the form

***<***VHDLRecordBaseType***>: <***VHDLRecordFieldType***> <***VHDLRecordFieldName***>; -- <***Register Function***>***

A line of the form

***<***VHDLRecordBaseType***> => <***VHDLRecordBaseType***>, -- <***Register Function***>***

Is also generated in the instantiation file, and also a line of the form

***signal <***VHDLRecordBaseType***> : <***VHDLRecordFieldName***>; -- <***Register Function***>***

is written to the top-level signal list file.

This spreadsheet entry does indeed save some typing and eliminates typographical errors. Because the generation of these lines puts semicolons and/or commas at the end, there is a user warning that if an #EntityDeclaration line is the last line in the spreadsheet, VHDL syntax errors may occur.

# Understanding the flow of the VBA code

 The VBA code is basically just one big FOR loop that searches all the rows of the spreadsheet from the declared start row to the last occupied row.

* The VBA looks at the value of cell B3 to find the starting row. A formula in that cell looks for the string ***%HDR*** in column A and puts the row number with the first instance of that key string, *plus one,* into cell B3. This is read by the VBA into variable *StartRow.*
* The VBA looks at the value of cell B2 to find the last row to process. A formula in the cell looks for the string ***#END\_OF\_DATA*** in column A, and the cell is filled with that row number *minus one.* The value of this cell is ready by the VBA into variable *StopRow.*
* The VBA code then uses a counter named *inrow* that runs from *StartRow* to *StopRow* in a big FOR loop. Each row is processed exactly once.

## Initial row processing

 Each row is read by a subroutine **ReadAllControlCells** that reads all cells from columns A through Z, with the exception of column F. Column F was labeled “Access Type” and originally it was thought this might be needed for EPICS conversions, but later on it was abandoned. This row is now vestigial and unused. Some minor pre-processing of data occurs to the data in columns B and Z.

* Column B (labeled “Register Address”) is understood to be the *relative address from some base* of the actual register. Cell E4 (with note “magic address offset insertion string for registers”) should contain the *base* address value. When processing all lines of the spreadsheet the address generated for any C or EPICS code will be the *sum* of the Column B value and the cell E4 value.
* VHDL register initial value strings in column Z are **always** understood to be hexadecimal numbers but it is understood that humans entering data into the spreadsheet may be careless. Thus the data value read from column Z is checked for a leading “x”, “X”, “0x” or “0X”, and if those are found they are stripped. Thus both careless and anal data entries are rendered identical for later processing.

## Decision Tree of main loop

 The main loop, after reading in a row’s worth of data, looks at the data read from column A. If the column A data is not null, a series of If…Then…Else statements process all the different keywords that are defined for use in column A. If the column A data doesn’t exactly match any of the keywords but is non-null, the row is treated as human commentary and passed to output files as a comment line.

 If, however, column A is null, then the data is understood to be either a **whole register line** or a **bitfield descriptor line**. The rule to define these two types whether column B (RegisterAddress) is blank or not.

* A line with column A blank and column B blank is processed as a **bitfield descriptor.**
* A line with column A blank and column B non-blank is processed as a **whole register.**

### Bitfields vs. whole registers processing rules in different output languages

 ‘C’ output files and VHDL output files are structured in ways that demand both pre- and post- processing output for registers. For example, the auto-generated register design in VHDL has to start with the syntax of an entity declaration, then list all the registers, then close the entity declaration. Similarly a ‘C’ struct defining registers would have to start with the typedef statement and curly bracket, then have the lines defining the elements of the struct (the registers), and then have a closing curly bracket. If the VBA code also has to develop unions in the ‘C’ output to map all the fields against the entirety of the register, this requires additional pre- and post-processing. These all combine to require a rigid spreadsheet format and internal flag variables that are set and cleared as the main FOR loop parses field lines and whole-register lines.

 Because of this the spreadsheet format is fixed with all field definition lines for the same register in a contiguous group, and the whole-register line on the line immediately following the field group.

 EPICS, on the other hand, works differently. EPICS generates separate and distinct process variables (PVs) for every field, and may or may not generate PVs for the whole registers depending upon the way the designers set up the database. Thus, every line in the spreadsheet is uniquely processed with no crossover between line types. Because some EPICS designers want whole-register information (e.g. Digital Gammasphere) and others don’t (e.g. GRETA) there needs to be a flag to control whether whole-register lines generate EPICS output. This flag is found in cell E5 of the spreadsheet.

# Subroutines and purposes thereof

* Initial setup prior to reading the spreadsheet:
	+ OpenAllFiles (no further calls)
	+ GenerateRegisterFileHeader\_A (no further calls)
	+ *GeneratePackageFileHeader\_A* (no further calls)
* Main loop calls based upon special codes in column A:
	+ *ReadAllControlCells* (no further calls)
	+ *GeneratePackageFileHeader\_B* (no further calls)
	+ GenerateRegisterFileHeader\_B (no further calls)
	+ WriteArrayOfRegisters (no further calls)
	+ WriteBlankRecordDefStart (no further calls)
	+ WriteBlankRecordDefEnd (no further calls)
	+ WriteRecordFieldLine (no further calls)
	+ WriteArrayDefinition (no further calls)
	+ WriteManualEntityDeclaration (no further calls)
	+ WriteLibraryDefinition (no further calls)
	+ WriteLibraryUseCase (no further calls)
* Main loop calls based upon processing while-register lines:
	+ WriteFullRegisterDefinitionLine
		- WriteEntityDeclarationLine (no further calls)
	+ WriteBitfieldDefinitionLine
		- GenerateNonRecordRegInLine (no further calls)
		- GenerateRecordFormatRegInLine (no further calls)
		- GenerateNonRecordRegOutLine (no further calls)
		- GenerateRecordFormatRegOutLine (no further calls)
* End of file processing:
	+ CloseAndClean
		- *GeneratePackageFileTrailer* (no further calls)
	+ *CopyInstantiation* (no further calls)
	+ *CopySignalDefs* (no further calls)
* Totally Separate from the main loop:
	+ CreateWordDocumentWithTable
	+ Fill\_table\_entry (empty subroutine)
	+ SaveAndCloseWordDocument
	+ Check\_python
1. If later we find that Python doesn’t process this field, VBA can and should use it to define initial values of registers. [↑](#footnote-ref-1)