

The Use of Legacy Code in GEDAE

M. J. T. Alphey
BAE SYSTEMS Avionics
Sensor Systems Division,
Crewe Toll House, Ferry Road,
Edinburgh EH5 2XS, U.K.
(Tel. +44 131 343 4273)

marcus.alphey@baesystems.com

ABSTRACT

When embarking on a new project, it is often desirable to be able to make use of existing legacy software. This can provide a stable code base from which to start – a set of known, tested core building blocks at the heart of the application, upon which new or improved features can be built. It can also drastically reduce the effort required in reaching a particular level of functionality or performance.

In general, integration of legacy code with a GEDAE model requires a degree of forethought and planning. Some of the design decisions embodied by the legacy code may impact the current design, and possibly constrain the usefulness of some of GEDAE's capabilities.

Depending on the styles and standards to which the legacy code has been written, some degree of manual pre-processing of the files may be necessary to make them suitable for integration. Attention must be paid to issues such as the use of global data, particularly when the processing is to be distributed. Features such as application specific data structures also require special consideration, and hardware specific functionality should be isolated so that it can be easily updated should the target hardware change.

Additional automated pre-processing, such as vectorisation, may also be carried out to improve the efficiency of legacy code. The implications of all pre-processing must be considered as part of the decision to use legacy code in the first place, since any such changes may affect the behaviour of the code and hence its reliability.

Within a GEDAE model, the interface with legacy code will be within primitive boxes rather than flowgraphs. Calling the legacy code may not always be as trivial a step as it would appear – certain steps may need to be taken to set up the legacy code prior to executing the desired function call. For example, it may be necessary to convert a box's input data from its GEDAE format into something suitable for passing to the legacy code prior to calling the desired function. A similar conversion may be required on the data returned from the legacy code before passing it on to the next box in the graph. These steps can usually be incorporated by putting a 'wrapper' around the legacy code.

This paper discusses the issues associated with the use of legacy code within GEDAE. It considers the various coding features that may be used by legacy code and the implications of trying to incorporate these features into GEDAE. Experiences of issues encountered and how they were addressed during the development of the Captor Tranche 2 programme – a radar project implemented in GEDAE that makes use of legacy C code – are described.

Comment is passed on enhanced or additional features of GEDAE that would improve its support of legacy code, such as optimising compilers and better support for archiving legacy code dependencies.

Keywords: GEDAE, legacy code, compilation, source, include, library, testing

1. INTRODUCTION

Software development can be a long and expensive process. One way of helping to reduce the amount of work that must be done is to make use of suitable existing software components from other projects. There are a number of factors that should be considered when deciding whether or not to make use of any such legacy code. These include assessing the amount of work that will be required in order to integrate the legacy code with that of the current project, and weighing this against the benefits that will be gained from doing so.

An additional benefit of using legacy code, above and beyond the saving in development time, includes the stable, tested code base it can offer. This can simplify the surrounding development by providing key components from which new functionality can be rapidly constructed. Effort can then be focussed on the new features of the software knowing that the legacy components have been thoroughly tested.

GEDAE supports the integration of legacy code through the compilation and linking of external source code. This allows primitive boxes to call the legacy functions directly. While there are no explicit constraints imposed by GEDAE beyond the availability of a C-compatible interface to the legacy functions, there are certain points that should be considered when deciding on the viability of using legacy code in conjunction with GEDAE.

2. LEGACY CODE CONSIDERATIONS

Many of the issues that may affect the level of work required to integrate legacy code with GEDAE relate to the standard and style in which the legacy code was written, and the way in which it was structured.

Some of the key issues are discussed below. The legacy source language considered is C, although the discussion is not restricted to this and the points addressed should have corresponding features in most programming languages.

2.1 Coding Standards

Boxes in the supplied GEDAE library are written to follow a particular set of coding standards. When developing custom GEDAE boxes, the developer must accept and adopt some of these standards in order to comply with the parsing rules used by GEDAE. However, within the various Method sections, the developer may follow a different set of standards with regard to the naming and layout conventions for the code. These may also be different to those used when the legacy code was originally written. Care must be taken that none of these standards permit names or constructs that conflict with the parsing rules, although this is unlikely. Very few features of C are not permissible, with the exception of bit-field selection, and most naming conventions will not directly clash.

There is, however, a possibility of loss of clarity due to differences in the coding standards. Care must be taken that the developers of the GEDAE model appreciate the naming and typographical conventions used in all of the standards concerned. They should add clarifying comments to GEDAE model components when any GEDAE conventions must be broken to make use of legacy components, be they names of data components or processing functions. Such clashes will

normally concern the declaration of parameter inputs to be assigned legacy program constant values or the capitalisation of stream names that output them, for example.

2.2 Global Data

Due to variations in the abilities of different compilers to reconcile multiply-defined symbols, care must be taken when making use of global data. Such data cannot be instantiated in header files that may be included by many boxes, but must instead be declared externally and instantiation moved to a source file. The corresponding object file can then be linked into the GEDAE executable at compile time, thereby allowing all of the legacy code in primitive boxes access to the global data.

Furthermore, since there is a possibility that different parts of legacy code that all require access to the global data may be distributed across multiple processors, consideration must be given to the distribution and control of access to this data. This particular problem is not one directly imposed by GEDAE – rather it is something that must be considered in any distributed processing environment. However, since GEDAE offers the option of distributing components of what might originally have been executed on a single processor, the issue must be addressed.

2.3 Program Constants

The issues associated with the use of legacy program constants in a GEDAE model are mainly concerned with ensuring access to the unique constant definition throughout all of the boxes in which it is to be used. If the constant is defined in a header file, that file should make full use of pre-processor conditional compilation macros to avoid multiple definitions of the constant.

Program constants may have been used in legacy code to set up explicit hardware addressing. Unless the original target hardware is still being used and explicit memory-mapping cannot be avoided, these addresses will now be wrong. In most cases, the memory required will now be allocated by GEDAE at scheduling, and as a result the use of the program constant may no longer be appropriate unless it can be redefined as a relative offset.

2.4 Macro Definitions

Macro definitions can be used within GEDAE primitive code just as they would normally have been used in the legacy application. However, it should be checked whether or not the macros reference any global data or program constants and if so, whether they must be redefined in light of the points raised in Sections 2.2 and 2.3.

2.5 Datatypes and Data Structures

The use of legacy application-specific datatypes and data structures is perfectly feasible within a GEDAE model, simply by referencing the appropriate type or structure definition and explicitly referencing structure fields or elements. Any processing of the legacy data structures will have to be carried out within a primitive box, unless the appropriate data structure support boxes have been created to split the compound data type into its constituent parts. (See [1] for

more details of how the generation of such boxes can be automated.)

Explicit type casting and type conversion functions can be used to accommodate the use of simple datatypes not directly supported by GEDAE.

Legacy structures that make use of bit-field selection will need to be modified, as GEDAE's primitive box parser does not support this notation. Consequently, any structure members that make use of this feature will instead have to use a whole integer to represent their permissible values – this change must be propagated to any other areas of the legacy code that may be impacted.

Note that due to the implementation of streams in GEDAE, access to members of a stream token data structure will need to make use of the structure pointer de-referencing operator (`->`).

2.6 Functions

In order that legacy functions can be used within a GEDAE primitive, they must have a C-based API. It is also very important to ensure that all functions are correctly prototyped. Failure to do so can lead to confusing results when running the model, since some default prototypes will conflict with that intended.

Functions that make use of global data will need to be checked to ensure that they will still work correctly once the comments on the use of global data given in Section 2.2 have been applied. In some cases, it may be necessary to modify these functions to take what was formerly the global data as a function argument.

Additionally, functions that explicitly allocate or free dynamic memory should be carefully examined to see whether they could be modified to move this memory management under the control of the GEDAE scheduler. Unless it must be done separately for each invocation, it may also be favourable to have any allocation carried out during the Reset stage of the graph execution, and freeing performed by the Destroy Method. These recommendations are not crucial to the correct operation of the model, but may improve efficiency.

3. PORTABILITY

When creating a GEDAE model it is desirable to make the code truly portable, to fit in with the GEDAE programming ethos and to ease its porting to any other platform. Consequently, any hardware-specific functionality in legacy code must be isolated and either removed (if it can be replaced by equivalent processing) or made generic. Note that this task of making the code generic can be achieved by using the GEDAE technique of writing a stub function with a number of different hardware-specific implementations and putting them in the appropriate sub-directory of the source embeddable directory. GEDAE will then link in the appropriate variant for the specified hardware at compile time.

This approach may require that a new variant is written for each new platform to be used, but this may be an easier or more optimal solution to generalising the functionality.

The points pertaining to explicit memory addressing discussed in Section 2.3 should also be considered as part of the assessment of portability.

3.1 Byte Ordering and Alignment

Another issue that must be borne in mind is the possibility of different target platforms using different byte ordering – whether they are big-endian or little-endian. This can be an issue for code that explicitly references particular bytes within longer data-words and for the use or interpretation of binary data – be it input or output data, or memory images – that has been created on one platform and moved to another whose byte ordering is different.

This can make debugging or comparison with output from the original code more difficult, since it means that direct memory dumps taken for these (or other) purposes cannot be used without first processing them to account for the current platform's ordering. Different byte alignment on different platforms may also cause disparities in such memory dumps and must again be compensated for before attempting any comparisons.

4. BUILDING LEGACY SUPPORT INTO GEDAE

Having considered the legacy code itself and made and tested any necessary changes, the next step is to make this available to the primitive boxes. This is achieved by compiling the legacy code modules, and linking them with the GEDAE executable. GEDAE provides a user-configurable mechanism for doing this, via customisations to two files in the user's GEDAE directory: `makeGEDAE` and `Personal_Obj_List`. When compiling code for target processors, the processor specific `Personal_Emb_Obj_list` and `runtime_make_info` files will also need to be updated correspondingly.

4.1 Compilation Flags

The setting of compilation flags in the `makeGEDAE` script permits any conditionally processed legacy code to be switched in or out appropriately. This can be particularly useful for enabling debugging diagnostic message output across the whole of the legacy code or for selecting hardware-specific sections of code. However, care should also be taken to ensure that the default compilation flags do not lead to problems with the compiler's treatment of the legacy code. Code can behave differently under different levels of optimisation or debugging, and the legacy code must either be thoroughly tested under the default GEDAE flags, or the flags should be changed to provide the default legacy settings.

It should be noted that conditional processing directives can also be switched on or off for specific GEDAE primitives or for files that they include by setting the appropriate flags in the Include Method of the primitive itself – this allows different boxes to make use of different sections of the same legacy code, if so required.

Include and library paths for legacy code can also be configured here, to avoid having to have all of the legacy code in the source and include embeddable directories.

4.2 Source and Include Files

Use of source and include files when building the GEDAE executable is normally fairly straightforward once all of the required files have been identified and providing that the

aforementioned considerations have been taken into account. Conditional processing directives can be used to prevent multiple inclusion or to omit parts of the code that are not required – however, care must be taken when inserting any such new directives that the code excluded is not unknowingly required elsewhere in other legacy files.

Problems can arise, however, if a particular directory structure is assumed by the legacy code files, since this structure must generally be flattened when the files are put into the GEDAE source and include embeddable directories. This is due to the make rules from which the GEDAE executable is built failing to find a match for the files in the nested directories. In this event, it may be easier to compile the code into a library outside of GEDAE and then link in this library when building the executable.

An organisational problem related to the use of legacy source and include files is that they are not included in the list of dependencies created and archived by GEDAE's FlowGraph Utility (FGU). While they can be added manually, this can be a non-trivial task if only the required files are to be added – it is, however, simple enough to add the entire contents of the two directories. This may be misleading, though, since not all of the files thus recorded are strictly necessary for the project being archived.

The process of identifying which legacy files are required by which project can be simplified by assigning groups of object files to different variables in the `Personal_Obj_List`, and then creating the `PERSONAL_EMBED_FILES` variable from a list of these group variables. Some files may be commonly required across projects and it may be advantageous to group these separately. It may be further desirable to allow primitive boxes to explicitly list their dependencies and have these collected by the FGU when the primitive is archived.

4.3 Libraries

When the desired functionality from the legacy code is distributed over a large number of files, or the directory structure of these files is important and cannot be flattened, it may be easier to compile the required functions into a library and link this in with the GEDAE executable instead. This can greatly simplify and reduce the contents of the source and include directories, as the files can now be stored elsewhere. All that matters is that the library file is in the appropriate location (specified in the `Personal_Obj_List` and the `makeGEDAE` script) at the time of building the GEDAE executable. Adopting this solution does, however, move the library maintenance issue outwith the control of GEDAE, requiring the developer to update the library file whenever any of its constituent components have been modified.

It is also possible that the required legacy functionality already exists as a fixed library in the appropriate format. In this case, the source code for the library is not required at all, and there will be no maintenance issues since the contents of the library are fixed.

5. USE OF LEGACY CODE IN GEDAE BOXES

When using legacy code in a GEDAE model, almost all of the interaction with the legacy interface will be from within primitive boxes. The only aspects of the legacy code likely to appear on a flowgraph canvas are local data declarations that duplicate a legacy program constant. Even then, this would normally only be used to pass that value into an existing GEDAE library box. Any such duplication should be commented to draw attention to the fact that this value may need to be updated if the original legacy code value ever changes.

GEDAE typedef constructor and destructor boxes may also appear on the canvas, but these are really primitives that have a different graphical representation – they are only a convenience to allow multiple flows of data to be represented by a single line on the canvas.

All of the real legacy processing will be called from within the various Method sections of a primitive box.

5.1 Primitives

In many cases, primitives that call legacy functions can follow a fairly simple pattern. This consists of a number of steps that, in most cases, should be contained within the granularity loop of the box's Apply Method. First, the input data must be converted into a suitable format for passing to the legacy function. Then, the legacy code itself is called, and finally any data returned from the function call is converted back into an appropriate format for passing out of the GEDAE box and onto the next downstream box.

Since the call to the legacy code is normally the only real processing that is being performed by the box, the remainder of the code in the Apply Method – the data conversion on input and output, and even the granularity loop itself – can be considered a wrapper.

The type conversions that go on around the calls to the legacy code should make use of explicit casting and type-conversion functions where necessary. These functions should also be linked into the GEDAE executable at build time. In the event that multiple legacy functions are called from within one primitive box, any data returned from a function that is to be used by a later function need not be reconverted back to a GEDAE-compatible format unless it is required to be output. It is also possible to declare local variables of the legacy datatypes, if necessary, for holding interim results.

In almost all cases, the box will only be able to fire with a granularity of one, as the legacy code is very unlikely to have been written in a fashion that supports anything more than this. However, it is good practice to include a granularity loop and consider ways that the legacy code might be improved to make use of it.

Some boxes may also require to perform particular set-up tasks prior to the desired legacy functions being called, *e.g.* setting up access to memory allocated by functions elsewhere in the legacy code. Whatever the set-up, this should be in the box's Reset Method wherever possible, if this can be achieved by a one-time configuration. Otherwise the set-up should be carried out at the start of the Apply Method, before the

granularity loop, unless this set up is dependent on the box's input data. Similarly, any tidying-up routines should be called in a box's Destroy Method wherever possible.

6. MODELLING CONSIDERATIONS

Having considered all of the issues with the compilation and use of the legacy code itself, there are also some constraints that the use of legacy code may impose on the modelling of the processing within GEDAE that should be noted. These are due to the ways in which GEDAE creates and controls the schedules that run the processing on the various data flows.

6.1 Dataflow Sizing

One area of difficulty that may be encountered is the desire to size data flows of vectors or matrices from values that may dynamically change, and which are supplied by legacy code functions. Data sizing parameters in GEDAE must be fixed at compile time, and hence the optimal way of modelling of such aspects is not always apparent. In this situation, there are a couple of alternative strategies that can be employed.

The first is to make the vectors and matrices into variable vectors and variable matrices. These datatypes are slightly less efficient to process than their fixed-size counterparts due to the overhead of invoking the dynamic scheduler and the need for more out-of-place operations and memory copies. It also means that the maximal sizes of the data sets must be known in advance and the actual sizes must be calculated separately when resizing the data to ensure that the correct amount of data is processed. Use of the over-specified variable matrix options helps make this solution more memory-efficient.

The second alternative is to design the processing using fixed-size vectors and matrices and then use GEDAE's enumerated scheduling capability to route the data to the appropriately-sized instance of the graph. This option is best used when the number of possible configurations for the data is relatively small. Use of exclusivity, while possibly impacting the throughput of the model, will reduce the memory requirements thereby keeping this approach memory-efficient.

6.2 Parameter Evaluation

Another processing consideration that should be noted is that boxes that carry out non-data-flow parameter-based evaluation cannot be mapped to embedded processors. This can sometimes lead to problems if any such legacy-related evaluation is to be carried out to provide a parameter input to a standard GEDAE box, as there is no other way of getting the necessary values to the box. In some cases, the processing can simply be re-written to use stream-based processing instead but this will slightly increase the processor loading.

For custom-written boxes, parameter evaluation can simply be performed in the box requiring the value after first including the necessary legacy definition of the required parameters – normally program constants.

6.3 Processor Loading

One disadvantage of calling individual functions that run large blocks of legacy code from within a primitive is that this processing cannot be split up and distributed across multiple

processors should the loading on any one processor prove to be too high. Depending on the structuring of the legacy code, it may instead be possible to put each of the sub-functions into a separate primitive and have them pass any necessary intermediate data between them. In this way, the processing can be more easily balanced, should this prove necessary. However, this will incur a slightly higher overhead due to the separate calls to the legacy functions from different points in the GEDAE schedule.

7. TESTING

As in any project, thorough testing is very important when integrating legacy code into a newly-developed model. It is important to realise that although the original legacy code itself should be as reliable as ever, any changes that were required must be tested to ensure that they have not affected the expected behaviour of the code. It is also necessary to confirm that the conversions being carried out to convert data to or from a format suitable for the legacy code are producing the anticipated values.

7.1 Debugging

Inserting conditionally included debugging diagnostic messages into primitives that contain legacy code can be extremely helpful in tracking down where problems occur if GEDAE crashes while running the model. This can help show whether or not the correct values are being passed to or returned from the legacy code and whether or not any conversions are being carried out correctly. Progress statements can also be useful in showing how far through a primitive is running before a crash.

Similar debugging messages can also be inserted into the legacy code if necessary, surrounded by pre-processor conditional processing directives to allow them to be switched out once the box or model is working correctly.

7.2 Comparison

Once the GEDAE model is running, it can be very helpful to perform a comparison of the primitives' output with the output of the original legacy code. This can confirm whether or not the code has been correctly integrated into the primitive and thus validate its behaviour.

In the case of a re-implementation, comparisons can also be carried out at a higher level, to check that the outputs from a particular graph or section of graph agree with any available legacy output. The overall output can also be checked, although care must be taken that the results are actually comparable. This can be an issue where the original platform and the new platform have different levels of, for example, floating point accuracy. Although the differences between the outputs of individual boxes may be minute and well within the desired tolerance, the accumulation of these differences may be enough to force a different route through some of the processing at a fairly high level, leading to a different output result. This can only be determined by careful examination.

8. USE ON CAPTOR TRANCHE 2

The CAPTOR programme is a joint European project to develop the multi-mode nose radar for the Eurofighter

Typhoon aircraft. The project is split into a number of phases :

- Tranche 1 (T1) developed a baseline radar which is now in production and being flown on a number of Eurofighter aircraft.
- Tranche 2 (T2) is a re-development of the system, primarily to support a COTS processor.

The T1 Signal Processor software is being re-developed in GEDAE at Tranche 2 to facilitate its porting to COTS hardware.

Since the processing itself will not change, it was decided to re-use those parts of the code that were written in C, targeted at a single processor, and portable. The remainder of the processing was to be re-implemented as GEDAE flowgraphs and primitives. This re-use reduces the complexity of the re-development work and associated documentation, and helps reduce the possibility of errors in interpretation and coding from being introduced during the re-implementation.

Tranche 1 legacy code assumed particular fixed locations in memory for certain data. Whilst it would have been possible to re-write these sections of code, it was decided that it would be easier to emulate the expected memory block. Consequently, the whole of this memory is allocated within certain primitives and loaded with the required values at the appropriate addresses prior to the legacy code routines being called. The contents of this memory are then extracted and passed on to the downstream primitives that require the information. With the introduction of state variables in GEDAE, this process should be simplified by allowing the memory block to be declared as a state variable and having GEDAE work out and control the access to it.

Support functions were written to convert between the Tranche 1 basic datatypes and their GEDAE equivalents for the different sizes of integers and for floating point data. As well as being essential to allow the correct interfacing between standard GEDAE boxes and legacy-encapsulating primitives, these also proved very useful for transforming the data to the same format as the data source when examining the output from the legacy processing boxes.

Development of the GEDAE model of the system is progressing well.

9. FUTURE WORK

The impact of high-level code-optimisers that perform operations such as vectorisation and parallelisation on legacy source code has yet to be evaluated for the Tranche 1 code, although this has been tried on other code in a separate study. This may offer an effective, automated way of pre-processing

the code to improve its performance in conjunction with its incorporation into GEDAE primitives.

While already well-suited to the task, further improvements to GEDAE could be developed that would facilitate use of legacy code :

- Better support for legacy source code debugging of primitives
- Improved dependency checking / support for relating source and include files to particular graphs to ensure that all of the required files are archived together
- Greater flexibility in organising the source and include directories, allowing files from different projects to be stored in separate directories

These would assist in the development, testing and organisational management of legacy-based modelling in GEDAE.

10. CONCLUSIONS

Legacy C code has been successfully integrated into GEDAE primitives, and used in the development of the Captor Tranche 2 programme. Coding and portability issues that should be considered when determining the amount of effort required in order to make use of legacy code have been discussed, as have key legacy-integration steps. Modelling constraints imposed by the use of legacy code have been highlighted and solutions proposed for commonly encountered problems. Strategies for improved testing, both during and after development are identified.

Experience of the integration of legacy code into the GEDAE model components for the Tranche 2 signal processing software have shown that this is a very worthwhile exercise. This has reduced the need to re-develop certain areas of complex processing while allowing the benefit of GEDAE's portability and optimisation features to be exploited.

11. ACKNOWLEDGEMENTS

The author acknowledges the support of Malcolm Wallace, Ross Johnston and Bobby Miller at SSD Crewe Toll who contributed to the development of this work.

12. REFERENCES

- [1] Alphey, M. "The Use of Autocoding in the Support of Application Specific Data Structures in GEDAE" in Proceedings of GUC2003 (Philadelphia PA, 2003).