I-Logix

UML based Model-Driven Development for C

By I-Logix

Abstract

The Unified Modeling Language (UML™) and Model-Driven Development (MDD) are rapidly becoming very hot topics and many companies are realizing the advantages made available by these technologies of better, standardized communication (UML) and the ability to reduce development time while producing higher quality designs by finding errors earlier in the process. The Object Oriented (OO) and C++ embedded community have rapidly adopted these technologies, but the spread into the C developer community has not been as fast. Up to now, the two main reasons that C Developers are reluctant to adopt UML were the unfamiliarity with OO constructs and the lack of ability to efficiently reuse legacy code. These challenges are being overcome through the introduction of natural C concepts such as files, functions and variables into the modeling language that enable the developer to think and work with the concepts that they are used to. Producing code structures the way C Developers always have makes it more intuitive and lowers the risk of change. In addition, developers can now integrate their existing code with the model by visualizing the external library or legacy code as part of a UML diagram without changing it, allowing them the advantages of graphical modeling while maintaining the full integrity of their legacy code. By being able to model using the same concepts they do today while being able to leverage the benefits of reusing code they have already written enables embedded C developers to gain the benefits of faster time to market and higher quality products that UML based MDD provides.

Table of Contents

UML based Model-Driven Development for C 1
Abstract 1
Table of Contents 2
Introduction 3
Functional Development Within UML 4
Reusing Legacy Code within a UML based MDD Environment 6
Validating the Design using the Model 7
Generating Code from the Model 8
Ensuring that the Model and the Code are always in Sync 8
Conclusion 8
Q & A 9
Can UML based MDD help when developing code for 8 & 16 bit microcontrollers? 9
What about connecting to a Real-Time Operating System? 9
What if we do not use a Real-Time Operating System? 9
Can I generate documentation from the model? 10
In UML, variables can be public or private, how can we achieve that in C? 10
What about initializing these variables? 10
How can we write our reset function? 10
What about handling interactions (relations) between Files? 10
How can one File call another File's function? 11
How could we add behavior to a File? 11
How can we generate code when a File has a statechart? 12
What about events, how can they be handled? 13
What about Activity diagrams? 14
Bring UML based MDD to life for the C Developer 15
Rhapsody in C/C++ 15
About I-Logix 15

Introduction

The UML is a standard designed and maintained by the Object Management Group (OMG) that allows software developers and systems engineers to graphically represent the requirements, specifications, structure and behavior of the systems they are designing. MDD technology extends UML allowing the achievement of unparalleled gains in productivity over traditional document driven approaches by enabling you to specify your systems and software design graphically, execute, to simulate and validate the system as you build it, and ultimately generate full production code from the model for the target system resulting in shorted development periods and higher quality designs. The Object Oriented (OO) and C++ embedded community have rapidly adopted these technologies, now, by adding concepts natural for the C developer and easily allowing for reuse of third party and legacy code, UML based MDD is ready to bring the same benefits to the C community.

One of the main reasons that C developers have not adopted UML is because they are used to function-based programming, and UML has always been designed for OO programming. This challenge is being overcome through the introduction of natural C concepts such as files, functions and variables into the modeling language. By allowing developers to program functionally by placing the files as well as functions and variables on diagrams in UML, C developers now have the ability to use the concepts they're used to while representing them in a UML diagram allowing C developers to design and think the way they normally do. This provides the advantages of graphical modeling without the need for changing to OO or C++. Furthermore, it also enables C developers to adopt UML based MDD with a much shorter ramp up time than switching to an OO approach or another target language.

It is not enough to only develop in a natural manner but the code produced from the model must be structured and have the same look and feel as typical hand code would. In addition there may be times when the code must be written or modified by hand in order to facilitate debugging, ensure proper interaction with the hardware or to achieve maximum performance. Thus, for C developers to extract the maximum benefit from MDD, they must be able to program and edit at either the code or model level while ensuring that the model and the code always remain in sync. By providing model/code synchronization and producing structured code further lowers risk by allowing UML and MDD to fit into your existing processes and environments. No need to change your current coding standards, compiler, debugger or target hardware.

Another element of existing environments is legacy code. Legacy code can be used "as is" or incorporated into the model to be further developed using MDD. When the desire is to keep legacy code "as is" without any edits or modifications, code visualization can link to and display the code in diagrams as a part of the model while leaving the code untouched. If the desire is to use code as a starting point with modifications and enhancements, simply reverse engineer the code into the model and edit at both the model and code level to receive the full benefits of MDD. These same methods are

also used to effectively incorporate code from external libraries or another tool into your model.

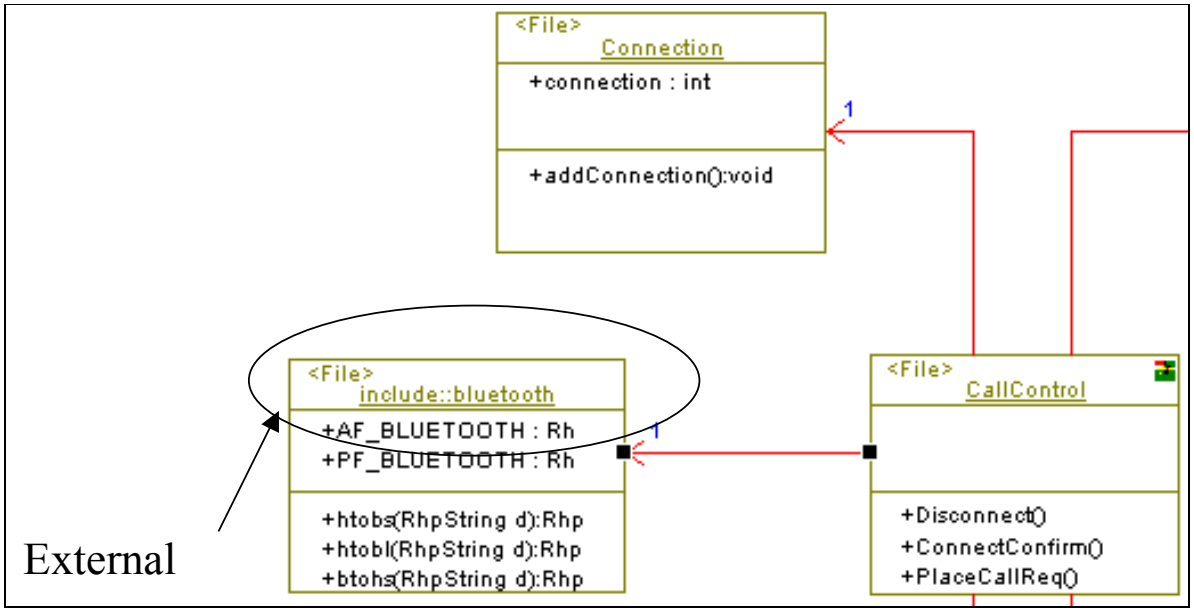

Figure 1: A structure diagram that contains three files, the variables defined in each file and the functions they contain. In other words it is a graphical view of the C code.

Note in the diagram referenced above that Connection and CallControl have been modeled (created) directly within the MDD environment while bluetooth is legacy code that has been visualized (included) in the model. Also, the variable connection and the function addConnection have both been created within the file connection and the functions Disconnect, ConnectConfirm and PlaceCallReq have been created within the file CallControl.

Functional Development Within UML

Developers can program functionally with UML diagrams by using a UML element called a file, which is simply a graphical representation of a source file. This file is capable of containing all the elements that C developers are used to dealing with including variables, functions, types, etc. The file is added to the diagram and is used to partition the design into elements much in the same way a class is used to partition a program in Object Oriented programming.

The code generated from the model appears very similar to the structural coding styles C programmers are familiar with. Rhapsody just represents .c and .h files that have the same name and couples them together as one element on a diagram called a file. If you do not use the .c and .h file coupling in your code, then we can represent just a .c or .h

file individually on the diagram. This means that developers do not need to learn how to do OO design, but can just bring the concepts that they have always used into the next level of abstraction, the model. In essence using the concepts of files, variables and functions in the model enables C developers to graphically describe their program and generate WYSIWYG code from the graphics. In addition the C developer can now simulate the design at the graphical level on the host PC before going to the target to ensure the behavior and functionality are correct.

Let's take the example of a Timer File; this File could have the responsibility to keep track of time. It could have variables such as minutes and seconds, and perhaps functions such as reset and tick. The reset function could initialize the variables to zero, and the tick function could increment the time by one second. In the UML we can create a simple Structure Diagram such as the following that shows a single File called Timer that has variables minutes and seconds of type integer, as well as public functions *tic*k() and *reset*().

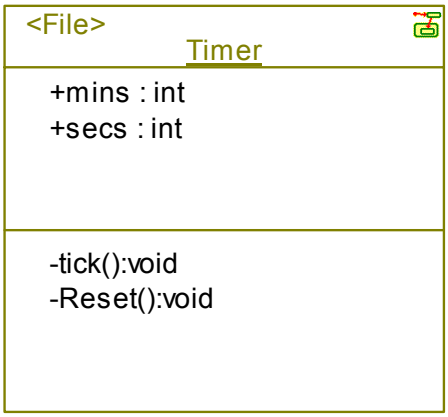

Figure 2: Structure Diagram

The C code for this file would look just like you would expect a typical C program to look like:

```
extern int mins;
extern int secs;

/*## operation Reset() */
void Reset();

/*## operation tick() */
void tick();
```

This code looks the same as what a C programmer might write, except it was generated from adding these elements into the diagram above.

Functions in one file can of course, communicate with functions contained in another file and they can also contain behavior defined by either a Statechart or an Activity Diagram. In addition, files and objects can both be used in the same model and files can be converted to objects. This enables developers who wish to migrate to an OO approach to do it at their own pace and doesn't force an all or nothing switch.

Reusing Legacy Code within a UML based MDD Environment

Most projects today have legacy code and we'll show how, with a powerful MDD tool, to easily reuse that code and receive the full benefits of MDD. One way to reuse your legacy code is to allow your MDD tool to represent your legacy code as diagrams through a unique process called code visualization. These diagrams are only representations of the code and all edits to these dynamically generated models take place on the code level only. The ability to generate these visualized diagrams allows legacy code to seamlessly integrate with a model-driven development environment while protecting the code's integrity. This method of using known good code without modifying it reduces the risk of having a bug and avoids wasted development and test time because no new code needs to be created or unit tested. The process simply creates a wrapper that acts as an interface or a façade to the legacy code, it can then be shown on diagrams and the legacy code can be linked in as a library. Here is an example of a model that has been visualized utilizing legacy Bluetooth code.

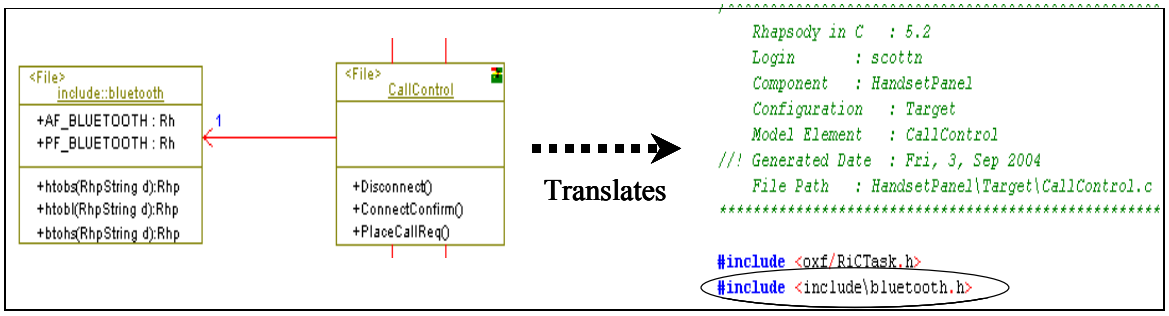

Figure 3: Generated code knows about external code automatically in Rhapsody.

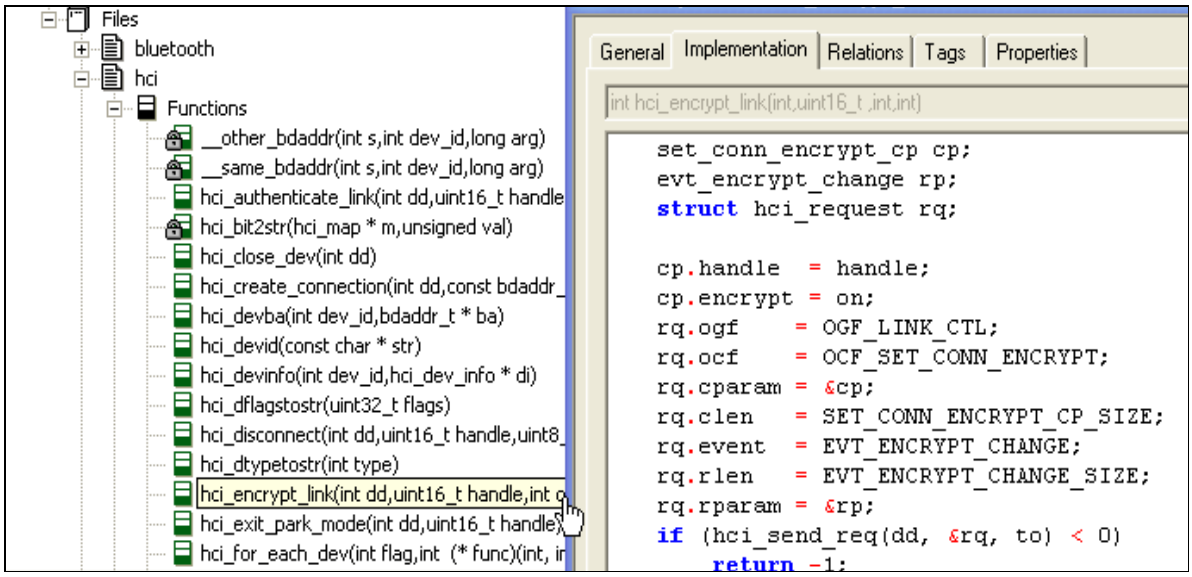

Figure 4: Visualization of all code artifacts in Rhapsody

Another way to reuse legacy IP is to simply reverse engineer the code right in to a model. Once part of the model, it is easy to modify, enhance, and further define the legacy system, and to gain all the benefits MDD including full production code generation, simulation to ensure errors are found and corrected early in the process and model-code associativity to enable changes to be made to the model or to the code directly while insuring the model and the code stay synchronized. The legacy IP becomes part of the modeled application and will be generated, updated refined, and tested within the MDD environment. This can be done all at once or piece-wise in an iterative reverse engineering of the system. Incremental testing of the new pieces of the system through model execution fosters a better understanding of how the legacy will fit within the modeled application and how your system performs as you incorporate it into MDD.

If you have some legacy code that you wish to keep as is and some that you would like to modify within your MDD environment. This is easily achieved, simply select the code you wish to keep as is to be visualized and select the code you wish to modify to be reverse engineered. You can also start off by visualizing the code as is and then if you decide you need to make changes you can then reverse engineer the code into the model.

Validating the Design using the Model

In a typical design, it is never clear that the model is correct until it has been executed. Technology is available today to allow graphical back animation or simulation, which basically means that code can be generated automatically from the model and instrumented so that, when executed, the model is animated. This means that the very

same diagrams used to describe the model can be used to validate the model. For example, the developer is able to see the value of the variables, see what each relation is set to, see what state each file is in, trace the functions calls between files on a sequence diagram, even step through an activity diagram. This animation can be done at any time during a project and allows the programmer to spend more time being highly productive doing design (the intellectual property) than spending time doing the tedious bookkeeping portions of coding. By being able to test and debug you design at both the model and code levels problems are detected early on and corrected when they are cheaper and easier to fix.

Generating Code from the Model

C code can be generated directly from the model; all the code that we have seen so far for the *Timer* File can be generated automatically. In fact for most models, between 65 and 90% of the code can be generated automatically. The remaining 10 to 35% is code that the programmer writes such as the bodies for the *tick* and *reset* functions. Code can be generated automatically for the dependencies, the association, files, functions, variables, Statecharts and Activity Diagrams, etc. The programmer just needs to specify the functions and actions on the statecharts.

Ensuring that the Model and the Code are always in Sync

Typically, programmers spend time creating a model and then generate the code. Later the code is modified to get it to work and nobody ever has the time or energy to update the model. As time goes on, the models get more and more out of sync with the code and become less and less useful. Again, technology is now available to ensure that any modifications to the code can be “round tripped” back into the model, ensuring that the code and the model are in sync at all times. This is so important during the maintenance phase as well as when features need to be added to a new version.

Conclusion

The introduction of natural C concepts such as files, functions and variables into the Unified Modeling Language now enable the C developer to receive all the benefits of Model-Driven Development while thinking and working the way they are used to. Through the process of visualization, it is now possible to incorporate legacy code into the development environment without changing a single line, enabling C developers to reuse there legacy code (IP), either as is or as a starting point. Outputting production quality structured code directly from the model further lowers risk by allowing UML and

MDD to fit into your existing processes further reducing the risk of adopting these technologies and providing an immense saving of development time. Many companies have already been doing development with UML based MDD and finding that they are reducing the development cycle by at least 30%.

Q & A

Can UML based MDD help when developing code for 8 & 16 bit microcontrollers?

The low cost, power consumption and wide array of on-board peripherals insure that 8 & 16 bit microcontrollers are here to stay. When developing code for these devices additional challenges are presented and with the right tool you can still benefit from UML based MDD. The ability to generate code that is structured and looks the way a typical C developer would write it combined with the ability to write or modify the code directly and have it automatically updated in the model enable the developer to create efficient code that can easily interact with the hardware and also include any special compiler directives.

What about connecting to a Real-Time Operating System?

Of course, we often want to be able to make a File (or a group of Files) run on a separate thread. We also want to be able to use Mutexes, Event flags, Semaphores, Message queues etc. We'd also like to be able to make our design independent of any specific RTOS, by using some kind of abstract operating system. This would allow us to use, say the Windows OS, so that we can rapidly validate our model on the PC. Once validated on the host, we could rapidly retarget for another RTOS to run on the target. An efficient way of handling this is to use a real-time framework. This framework (OXF) could be constructed so that it is easy to adapt it to another RTOS such as VxWorks, Integrity, pSOS, OSE, Nucleus, as well as a custom or proprietary OS.

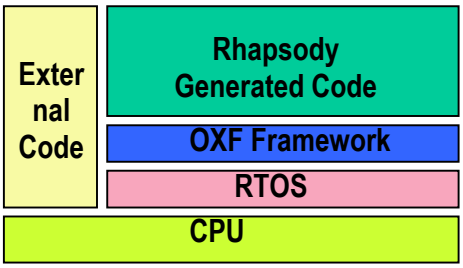

Figure 5: The OXF is used to map the generated code to many commercial RTOS'

What if we do not use a Real-Time Operating System?

This is often the case for embedded C developers. An Interrupt Driven Framework (IDF) can be your answer. This will provide a very low overhead solution and as long as

it was developed using a model within your tool then you should be able to easily decide if you would like to use static or dynamic memory allocation, remove any unused services, support host based execution and port it to any target environment that you chose.

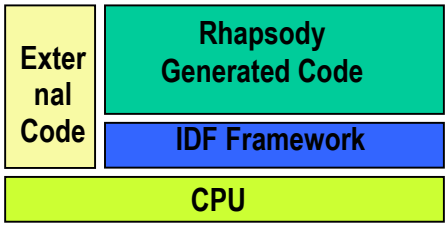

Figure 6: The IDF allows the generated code to run directly on your hardware.

Can I generate documentation from the model?

Of course, since the model contains everything, a report can be easily and automatically generated according to various standard templates, in word, HTML, framemaker, or your tool of choice.

In UML, variables can be public or private, how can we achieve that in C?

The variables that we have shown so far are also all *global*. A *local* variable can only be used within a File, so in C it makes sense to not declare this variable in the specification (.h) file. Rather, declare it in the implementation (.c) file and no longer have it declared as an extern.

What about initializing these variables?

A file can have an initialization functions or simply set the code equal to 0. By setting it in the tool you can get code like:

int mins = 0;
int secs = 0;

How can we write our reset function?

Just do it normally in the body.

mins = 0;
secs = 0;

What about handling interactions (relations) between Files?

A lot can be achieved by using one File but normally systems are composed of a number of Files that interact together. For two Files to communicate they need to be connected via some sort of relation. For example if we wanted our *Timer* File to call the show operation of a *Display* Files, then we would add a Dependency between the *Timer*

and the *Display* File. This will generate the include for Display inside the Header or C file on Timer, depending on how you set properties. It allows you to view your Includes graphically. The preferred way is to use associations not dependency, because this closely mimics UML modeling, and also provides an easy migration path between Files and Objects (if anyone wanted to move to an object oriented approach in the future).

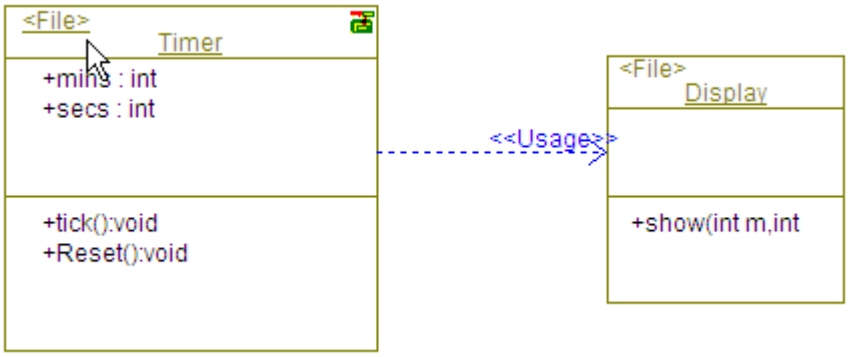

Figure 7: The arrow enables Display to be included in Timer

How can one File call another File's function?

We have seen that the relation has been coded as a simple include to the other File, so in our example, the*Timer* could call the *show* operation of the *Display* File directly.

How could we add behavior to a File?

In the UML, behavior of a File can be described by using a statechart (enhanced state diagram) or Activity Diagram (enhanced Flow Chart). For example, we could add a statechart to our *Timer* File to call the private *tick* function every 1000 milliseconds. The statechart would look like the one on the right: We could also add an "action on entry" to call the *show* function on the *Display* file. That way every second we will increment the time and display it.

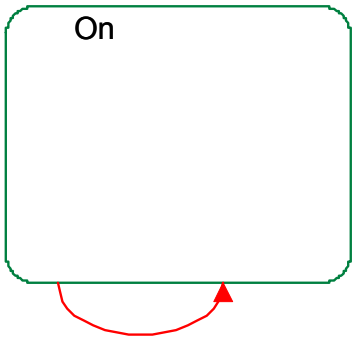

Figure 8: A simple statechart

How can we generate code when a File has a statechart?

This is getting more complicated and it is at this point that we really need to think about building some kind of real-time framework. We need to be able to have some mechanism for handling statecharts and some mechanism for handling timeouts.

```
enum Timer_Enum{ Timer_RiCNonState=0, Timer_Active=1, Timer_On=2, Timer_Off=3 };

struct Timer_t {
   RiCReactive ric_reactive;

   /***   Framework entries   ***/

   int rootState_subState;
   int rootState_active;
   int Active_subState;
};
```

This is automatically generated by Rhapsody.

This real-time framework could contain some reactive element that could be added to our Timer File to basically wait for events or timeouts to occur. Whenever an event or a timeout occurs, it could then call an operation to dispatch the event/timeout. For our example, the code could look like the following:

```
static void Timer_rootState_dispatchEvent(void * const void_me, short id) {
    switch (Timer.rootState) {
      case Timer_On:
      {
```

```
    if(id == Timeout_id)
       {
          Timer.rootState = Timer_On;
          RiCTask_schedTm(Timer.ric_reactive.1000)
       {
       break;
    };
    break;
   }
}
```

What about events, how can they be handled?

An event is generally asynchronous, but sometimes when we need the execution to be more deterministic, events can be synchronous. To see how events are handled, let's modify the Timer statechart to allow the Timer to be started, stopped and reset asynchronously via events. The resulting statechart will look as follows:

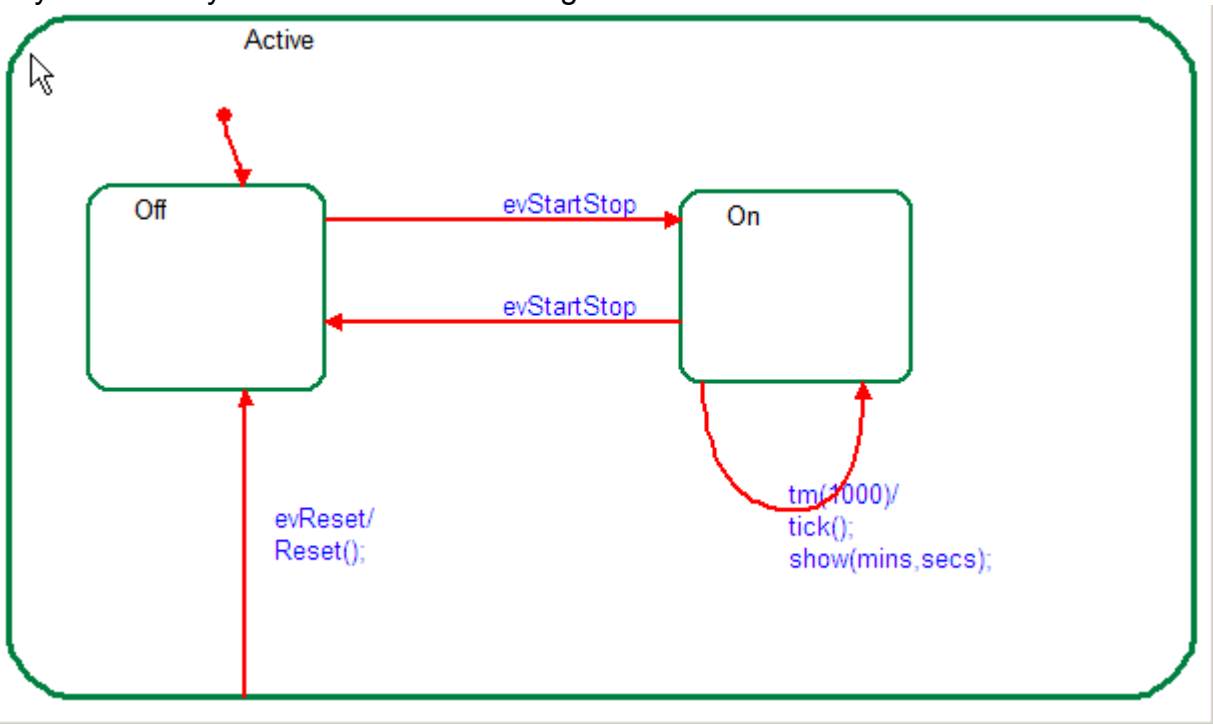

Figure 9: A statechart that can be used to represent the behavior of a file.

The *Timer* File will now initialize in the *Off* state and then wait for either the event *evStartStop* or *evReset*. On receiving the event *evStartStop*, the file will change from the *Off* state to the *On* state. In the *on* state, the *tick* operation will be executed every second. In this state if the event *evStartStop* is received then the file returns to the *off* state. In either state, if the event *evReset* occurs, then the *reset* function will be executed and the file will result in the *off* state.

Again, a framework needs to be created for handling events. To send an event, a macro could be created so that an event can be sent to a File. To send the event *evStartStop* to our *Timer* File, the following code could be deployed for another File such as a *Button* that has a relation to the *Timer* File.

```
CGEN(&Timer,evStartStop());
```

What about Activity diagrams?

Simple Activity diagrams or flowcharts can be attached to operations for describing the behavior of a function, so that complex functions can be captured graphically; code can be generated from these diagrams as well. Here is an example of such a diagram.

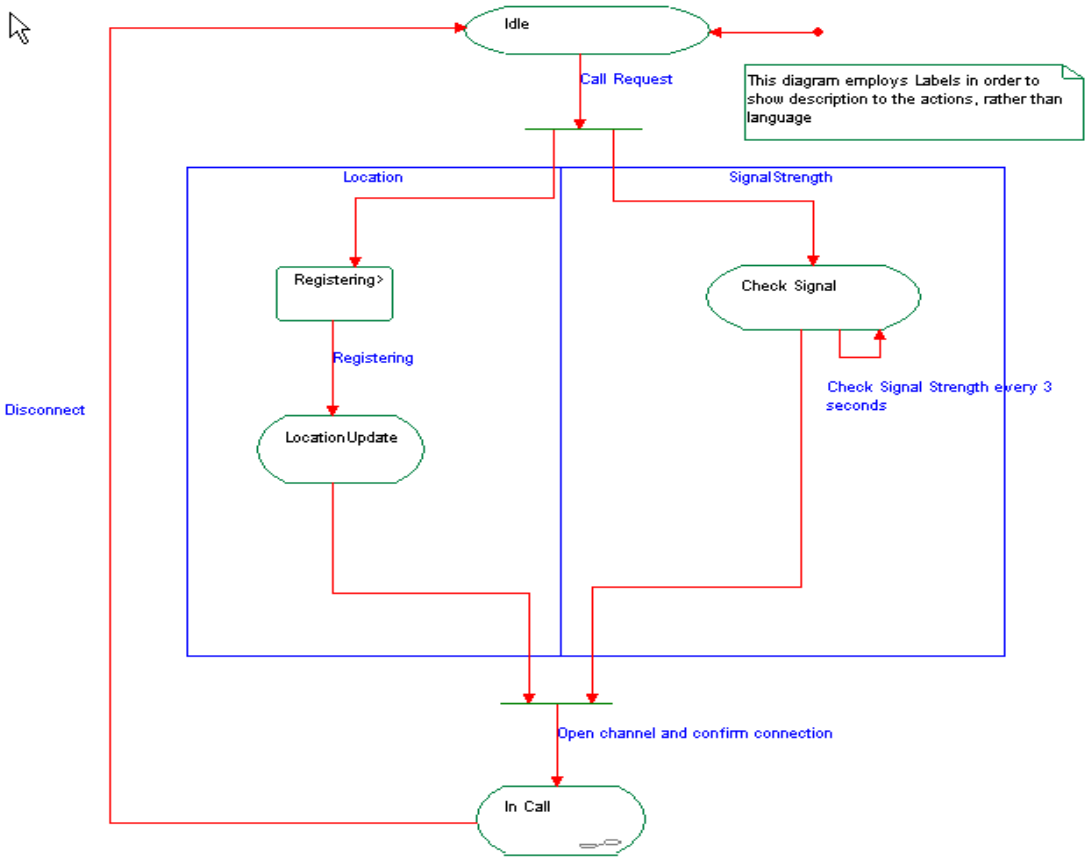

Figure 10: An activity diagram that can be used to represent the behavior of a file.

Code can be generated automatically, but the code for each individual activity must, of course, still be written by hand.

This paper was made possible by contributions from the following people:

Marty Bakal	I-Logix Senior Application Engineer
Rick Boldt	I-Logix Director of Product Marketing
Scott Niemann	I-Logix Rhapsody Product Marketing Manager
Mark Richardson	I-Logix Senior Application Engineer
Todd Szahun	I-Logix Marketing Communications Specialist

Bring UML based MDD to life for the C Developer

Now that you have read about how the C developer can benefit from UML based MDD, why not see it in action. Please stop by our website and view an on-demand webinar, or download Rhapsody for a 30 day trail.

Rhapsody in C/C++

I-Logix Rhapsody is an award-winning, UML-compliant, systems design, application development, and collaboration platform. Rhapsody is used by design and development teams to deliver real-time embedded applications in C, C++, Java, and Ada. Rhapsody uniquely combines a graphical UML programming paradigm with advanced systems design and analysis capabilities and seamlessly links with the target implementation language, resulting in a complete model-driven development environment, from requirements capture through analysis, design, implementation, and test. Operating on either an engineering workstation or PC, Rhapsody has the ability to "execute" graphical models and debug behavior at the design level, both at the host as well as the target, produce test vectors and generate production-quality code within a single development environment. Using Rhapsody, developers can maintain applications at the code level through graphical models. If changes are made to the model, the application code changes as well to correspond to the modifications; if the code is altered, the models are dynamically updated. The dynamic, bi-directional link between the code and the design documentation ensures that Rhapsody is ideally suited for all members of the design team. Rhapsody also incorporates automatic test generation and web-based model debugging to enable design for testability. Leveraging these innovative techniques via an intuitive user interface, Rhapsody dramatically boosts productivity and dramatically improves team collaboration, quality, and re-use leading to significantly compressed development cycles.

About I-Logix

Founded in 1987, I-Logix is the leading provider of Collaborative Model-Driven Development (MDD) solutions for systems design through software development focused on real-time embedded applications. These solutions allow engineers, operating in either small or very large teams, to graphically model the requirements, behavior, and functionality of embedded systems. The design is iteratively analyzed,

validated, and tested throughout the development process while automatically generated production quality code can be output in a variety of languages. I-Logix facilitates team collaboration through unique project and task management capabilities integrated into its UML based MDD solutions, enabling design review and inter-team participation from concept-to-code, regardless of where team members are located. I-Logix is headquartered in Andover, Massachusetts and has sales and support centers throughout North America, Europe and the Far East. For more information, please visit our website www.ilogix.com.