Developing embedded software using graphical programming

Björn Beckman

Master of Science Thesis
Stockholm, Sweden 2009
Developing embedded software using graphical programming

Björn Beckman

Master of Science Thesis MMK 2009:76 MDA 363
KTH Industrial Engineering and Management
Machine Design
SE-100 44 STOCKHOLM
Sammanfattning

Abstract

The increase of complexity in embedded programming is forcing the development of higher level programming languages. These languages not only consist of traditional text based programming but also with graphical programming languages such as the programming language G from LabVIEW which is developed by National Instruments and have been around since the mid eighties. However since 2004 this language has also been able to program embedded hardware, using the modules LabVIEW SDK and LabVIEW for ARM. This thesis shows how LabVIEW for ARM converts its graphical language G to C and then how C is flashed into the microprocessor. The thesis also takes up the pros and cons of using G for embedded development. The conclusion is that G is extremely useful for people without embedded knowledge that still want to develop embedded solutions. For customers with textual programming experience a time/cost analysis should be performed to see if G is more cost effective. The disadvantage is that LabVIEW SDK only can be used with 32bit processors and LabVIEW for ARM is even more restricted because it can only be used with ARM architecture processors. Another disadvantage comes into play if the user chooses to use a microprocessor which is not sold distributed from National Instruments because then a framework is needed to be constructed before usage, however this thesis shows in detail how such a framework can be developed.
# Table of Contents

Sammanfattning...................................................................................................................................................... 3
Abstract .................................................................................................................................................................. 4
List of abbreviations .......................................................................................................................................... 7
1: Introduction .................................................................................................................................................... 7
   Background .................................................................................................................................................. 7
   What is graphical programming ................................................................................................................. 7
Paradigm shift in embedded development ........................................................................................................ 8
   1.2 Purpose ................................................................................................................................................ 8
   1.3 Problem statement .............................................................................................................................. 8
   1.4 Boundaries .......................................................................................................................................... 8
2: Method .......................................................................................................................................................... 8
   2.1 Compiling the graphical language ...................................................................................................... 8
      2.1.1 G to C........................................................................................................................................ 8
      2.1.2 C to Assembler .......................................................................................................................... 14
      2.1.3 G to C to Assembler examples .................................................................................................. 15
      2.1.4 LabVIEW performance compared to C .................................................................................... 17
   2.2 LabVIEW Embedded ............................................................................................................................ 19
      2.2.1 LabVIEW Embedded Module for ARM .................................................................................. 20
   2.3 HM20 S-2 Module ................................................................................................................................ 20
      2.3.1 ATMEG AT91RM9200 ............................................................................................................ 20
      2.3.2 FPGA Spartan 3E ...................................................................................................................... 20
   2.4 LabVIEW training .................................................................................................................................. 20
   2.5 LabVIEW embedded for ARM training on evaluation boards ............................................................. 21
      2.5.1 EK-LM3S8962 .......................................................................................................................... 22
      2.5.2 MCB2300 .................................................................................................................................. 23
3: Testing .......................................................................................................................................................... 23
   3.1 Porting LabVIEW ARM to a processor ............................................................................................... 23
   3.2 Flashing the ARM processor .............................................................................................................. 24
   3.3 FPGA programming with LV ............................................................................................................. 25
   3.4 LCD testing ......................................................................................................................................... 25
1: Introduction

Background
For many years embedded development have been dominated by low level programming languages like Assembler and even more important C/C++. C gives the developer a very good control over the hardware and timing requisites. However, with the development of more and more advanced microprocessors and the rise of complexity in embedded designs the demands of programmers have increased and development times have risen dramatically [7]. One of the biggest problems is that few developers have sufficient knowledge about drivers, board support packages and bootloaders [8]. A solution to this problem is to use a higher level programming language like .NET or LabVIEW embedded for ARM (LV ARM).

LV ARM is the latest software from National Instruments (NI) and is focused on developing embedded applications for microprocessors with ARM architecture. Out of the box LV ARM supports two developments boards with ARM architecture, the Keil MCB2300 and the LM3S8962. NI Sweden is, however, very interested to see how LV ARM would work with any other development board that houses a microcontroller with ARM architecture, a Tier 2 device. This is where Prevas comes in to the picture. Prevas in Copenhagen have developed a development board, theHM20 S-2 ARM9/FPGA Platform, using the ATME AG91RM9200 ARM9 microprocessor combined with an Spartan 3E FPGA from Xilinx.

What is graphical programming
Graphical programming (GPL) or visual programming is any programming language that lets users specify programs by manipulating program elements graphically rather than by specifying them textually [25]. Most GPLs are based on the idea of "boxes and arrows," that is, boxes, circles or bubbles, treated as screen objects, connected by arrows, lines or arcs. There are currently more than 50 different types of visual programming languages [25]; one of them is the language G, which is developed by National Instruments. This thesis will mainly cover this language. G is a dataflow programming language which means that the program implement dataflow principles and architecture in contrast to imperative programming of traditional programming languages. Dataflow programs are generally represented very differently inside the computer as well. A traditional program is just what it seems, a series of instructions that run one after the other. A dataflow program might be implemented as a big hash table instead, with uniquely identified inputs as the keys, and pointers to the code as data. When any operation completes, the program scans down the list until it finds the first operation where all of the inputs are currently valid, and runs it. When that operation finishes, it will typically
put data into one or more outputs, thereby making some other operation become valid [34]. Execution is determined by the structure of a graphical block diagram (the LV-source code) on which the programmer connects different function-nodes by drawing wires. These wires propagate variables and any node can execute as soon as all its input data become available. Since this might be the case for multiple nodes simultaneously, G is inherently capable of parallel execution i.e. if there are two or more separate loops they will be executed on different CPU cores. G also gives the user powerful tools to create applications without writing any lines of text-based code, the user only drag and drop pre-built objects, express Vis, or low level functions to quickly and simply create user interfaces for the application. Then the system functionality can be specified by assembling block diagrams and is commonly used for data acquisition, instrument control and industrial automation on a variety of platforms including Windows, UNIX, Linux and Mac OS [26].

Paradigm shift in embedded development

Dataflow languages contrast with the majority of programming languages, which use the imperative programming model. In imperative programming the program is modeled as a series of operations, the data being effectively invisible [35]. This distinction may seem minor, but the paradigm shift is fairly dramatic, and allows dataflow languages to be spread out across multicore, multiprocessor systems for free.

1.2 Purpose

The purpose of this thesis is to find out how to control microprocessors with ARM architecture using LabVIEW embedded for ARM software. More specifically to analyze how the graphical language G from LabVIEW is compiled to C code and the pros and cons of using graphical embedded programming. This study will end in the programming and developing of a framework to a module using ARM architecture. This is because National Instruments want to be able to sell the software to developers who are using third party development boards and because Prevas wants a tool to develop their new platform.

1.3 Problem statement

How does graphical embedded programming work and what are the benefits of using this.

1.4 Boundaries

The boundary is that the theory will be written for all ARM processor but the actual framework will be made to the HM20 S-2 development module housing the ATMEAL AT91RM9200 microcontroller.

2: Method

2.1 Compiling the graphical language

LV ARM uses a two stage process when it converts its graphical language G to machine code which can be run on a microcontroller. First is compiles the code to C, which is the standard for embedded programming languages. After that the C code is being converted to Assembler using a third party compiler, the default compiler is from Keil and is called uVision, but other compilers can be used as well. However, the built in capability between LabVIEW and uVision gives the user additional work if he choose to use another compiler.

2.1.1 G to C

When coding Windows or Linux applications in LabVIEW the graphical code is instantaneous interpreted and compiled. This means that the programmer can see if his code is run-able or contains errors by just looking at the run-button, see figure 1. This feature is used when compiling to an embedded device as-well to all extent except the use of inline C-nodes and elemental I/O containing C-nodes which is checked for errors after the run-button have been pressed.
The way LabVIEW C Code Generator generates its G code from a LabVIEW block diagram to ANSI C is a standard process. When you build a block diagram into an embedded application, LabVIEW traverses the block diagram and generates simple C primitives, data types provided by a programming language as basic building blocks [28], if possible. For example, the LabVIEW C Code Generator converts While Loops to while() statements and converts the Add function to a simple C operation. However, a linear mapping is not possible for more complex functions i.e. all functions in LabVIEW cannot just be directly presented as a C function, so the LabVIEW C Code Generator uses the LabVIEW C Run-Time Library to handle more complicated code [27]. The definition of a runtime library is that it includes information of how the functions from the higher level language should be implemented to the compiled language [13], in this case the language C. The LabVIEW runtime library contains more than 750 functions [13] for handling of features like memory management and string formatting. The code that the LabVIEW C Code Generator creates first passes through a cross-compiler which add libraries with preexisting C code and the extra C files the user included and is then linked into an executable which can run on the embedded device.

On top of all that is written above LV ARM is using a well known Real-time operating system (RTOS) to give the C-code multitasking abilities, RTX. RTX is developed by Keil and is a real-time kernel for ARM7, ARM9, and Cortex-M3 devices [36]. It allows the creation of embedded applications that simultaneously perform multiple functions or tasks and it makes it possible to take advantage of the parallel workflow in LabVIEW for embedded applications. The use of a RTOS does not necessary produce code with higher throughput but instead a RTOS provides deterministic timing and scheduling abilities and the possibility to run processes parallel. The use of parallel processes in embedded systems is, of course, only a saying due to hardware constraints but a RTOS uses a Time-sharing design which switches tasks on a clock interrupt to give the illusion that a process or user has sole use of the machine [37].

In terms of performance, LabVIEW includes a compiler that produces native code for the CPU platform. The graphical code is translated into executable machine code by interpreting the syntax and by compilation. The LabVIEW syntax is strictly enforced during the editing process and compiled into the executable machine code when requested to run or upon saving. In the latter case, the executable and the source code are merged into a single file. The executable runs with the help of the LabVIEW run-time engine, which contains some precompiled code to perform common tasks that are defined by the G language. The run-time engine reduces compile time and also provides a consistent interface to various operating systems, graphic systems, hardware components, etc. The run-time environment makes the code portable across platforms. Generally, LV code can be slower than equivalent compiled C code, although the differences often lie more with the users programming optimization than inherent execution speed.

**Enumerating the processor targets**

An execution target list is always built when a VI is generated to C. The dynamic link library file TgtSupp.dll enumerates all targets to build the list. TgtSupp.dll supports the following C functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CountDevices(void)</td>
<td>Returns the number of discovered devices or sub targets.</td>
</tr>
<tr>
<td>DescriptorSize(void)</td>
<td>Returns the length of the longest target descriptor string.</td>
</tr>
<tr>
<td>IdSize(void)</td>
<td>Returns the length of the longest ID string.</td>
</tr>
<tr>
<td>EnumerateDevices(char* idBuf, char* descrBuf, unsigned long* kindList)</td>
<td>Enumerates all subtargets and returns a list of all targets.</td>
</tr>
</tbody>
</table>

* `idBuf` returns a list of ID strings.
* `descrBuf` returns a list of description strings.
unsigned long nDevices, int* ContainsSearchStrings)  *kindDevices returns a list of the kinds of devices. Returns 10 for all embedded targets. *ContainsSearchStrings is reserved for future use. Set to FALSE.

A VI Server (object-oriented, platform-independent technology that provides programmatic access to LabVIEW and LabVIEW applications) system call translates the VI hierarchy into C code. The term server is, however, a bit misleading because it has nothing to do with a server in a traditional sense, it's just a name for the API. Each VI in the VI hierarchy is translated into separate C files, which can later be viewed in uVision. The C files have the same name as the VI, with a .c file extension instead of a .vi file extension. This VI Server call also generates the following header files.

<table>
<thead>
<tr>
<th>Header File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LVForms.h</td>
<td>Contains declarations for the entry point functions for each VI.</td>
</tr>
<tr>
<td>LVGlobs.h</td>
<td>Contains declarations for global initialization functions and data type information used throughout the embedded application.</td>
</tr>
<tr>
<td>LVISRList.h</td>
<td>Contains a list of Interrupt Service Routines (ISRs).</td>
</tr>
<tr>
<td>LVFuncsUsed.h</td>
<td>Contains usage information to support source-level dead stripping.</td>
</tr>
<tr>
<td>debugtable.h</td>
<td>Contains lookup information for serial, TCP/IP, and CAN debugging.</td>
</tr>
</tbody>
</table>

The input to this VI Server call is a variant that contains attributes (name, value pairs) that determine how the LabVIEW C Code Generator generates the code. Any attributes you do not set use the default behavior. There are many attributes that determine how to generate the C code. Most of the attributes are optimizations that make the generated C code smaller and run faster.

**Code generation attributes**

As mentioned above user can, to some extent, choose how generation of the C code will be done. The benefit of this is that a program can be compiled different regarding to what kind of application the program is used for. There are many attributes that determine this generation of C code. Most of them are optimizations which make the code smaller and/or run faster.

**BigEndian**

*Type*  Boolean  
*Default*  False  
*Description*  Specifies if a platform is big endian or little endian. The LabVIEW C Code Generator must know if a platform is big endian or little endian in some cases to generate C code in byte form for compound data [27].

**DestinationFolder**

*Type*  Path  
*Default*  <empty path>  
*Description*  Indicates where you want the LabVIEW C Code Generator to save the generated C files. Unless the IncrementalBuild attribute is TRUE, LabVIEW overwrites any C files in the destination folder with the same name as VIs in the embedded VI [27].

**GenerateGuardCode**

*Type*  Boolean  
*Default*  True  
*Description*  Guard code prevents a user from making simple common mistakes that can cause an application to crash. For example, guard code can prevent dividing by zero and indexing out of range in an array. Guard code does make an application slightly larger and slower so it's a trade-off between performance and reliability. Set to FALSE to not generate guard code, which makes the code smaller and faster, but less safe and more likely to crash because of user mistakes [31]. The following LabVIEW functions use guard code:  
- Array index  
- Array replace  
- Compound arithmetic
• All floating-point arithmetic operations
• Quotient and remainder

**GenerateSerialOnly**
Type: Boolean
Default: False
Description:
The default generated C code has the same execution behavior as LabVIEW on the desktop. For example, parallel While Loops run in parallel. However, the LabVIEW C Code Generator generates code that uses cooperative multitasking in a single thread. Additional threads are used only by Timed Loops. The generated C code is difficult to read and does not resemble manually written code. [31].
Set to TRUE if you do not want to generate the cooperative multitasking code. The code is easier to read and usually runs much faster without the cooperative multitasking code. However, you lose the parallel execution behavior [31].

**UseStackVariables**
Type: Boolean
Default: False
Description:
Uses stack variables rather than heap variables to represent signals in the generated C code, which enables better C compiler optimizations and makes the C code faster. The reason for using stack variables as optimization is made on the basis that C programmers frequently use the stack. So, in other words, if there is a good optimizing compiler, it might speed up stack variables since that is the most common way to pass data between functions.

**ExpressionFolding**
Type: Boolean
Default: False
Description:
Generates better performing and more efficient code by collapsing groups of nodes into single expressions that are easily recognized by C compilers [27]. However, because ExpressionFolding eliminates some of the wires in the generated code, debugging the embedded VI is not viable.

**GenerateFunctionCalls**
Type: Boolean
Default: False
Description:
Set this attribute to TRUE to generate the interface to all VIs as C function calls without any default data initialization, which can reduce the code size by as much as 50% for a small VI. An error occurs if any input or output to any VI is unwired when the LabVIEW C Code Generator generates the C code [27].

**GenerateDebugInfo**
Type: Boolean
Default: False
Description:
Enables instrumented debugging and adds the extra code needed to debug over serial, Ethernet, or CAN protocols if you set this attribute to TRUE. The extra code contains function calls that update the application state and communicate the state to the host computer for display. Setting this attribute to TRUE usually results in a 25%–40% increase in code size [27].

**GenerateInlinePrintf**
Type: Boolean
Default: False
Description:
Indicates you want to use the C run-time library function printf if it is available for your target instead of the Format Into String function in LabVIEW. The printf function is usually smaller and faster than the LabVIEW C Code Generator run-time version for simple data types [27].
**GenerateIntegerOnly**  
**Type** Boolean  
**Default** False  
**Description** Generates C code that does not have any floating-point declarations or operations if the block diagram does not contain any floating-point data types. [27]

**GenerateInlinePrintf**  
**Type** Boolean  
**Default** False  
**Description** Indicates you want to use the C runtime library function printf if it is available for your target instead of the Format Into String function in LabVIEW. The printf function is usually smaller and faster than the LabVIEW C Code Generator runtime version for simple data types [27].

**GenerateLibraryInterface**  
**Type** Boolean  
**Default** False  
**Description** Generates additional code so that external, non-VI code can call the VIs as if the VIs are library functions. This attribute is supported only for strings, 1D arrays, and flat clusters [31]. The LabVIEW C Code Generator generates a C function interface for each VI, which configures the inputs and calls the normal LabVIEW VI interface. The C function name is the same as the VI name unless the VI name contains a disallowed character, such as a space, in the filename. Underscores replace disallowed characters. The VI behaves the same as a VI running under the My Computer target, or Windows, by using default data if any input or output is null. Finally, a header file is created that contains all of the function prototypes for the VIs that can be included where they are called [27].

**MemoryModel**  
**Type** Integer (0 = dynamic, 1 = static)  
**Default** 0  
**Description** Defines whether you are using a dynamic or static memory model.

- **Dynamic**—Memory is allocated and then freed as soon as possible, which keeps the memory use at any one time to a minimum. Memory management overhead might result in some jitter and reduced performance, but applications with a small memory model have the smallest footprint possible.
- **Static**—No memory allocation occurs while the application is running. All variables are declared as static in the generated C code and allocated prior to the main entry point. Static memory can provide good performance and low jitter, but static memory models might require more memory than other memory models. [27]

**Generating the fastest code**  
Use the following code generation options to generate the fastest code, but have the least amount of built-in error checks:  
- GenerateGuardCode=FALSE  
- GenerateSerialOnly=TRUE  
- UseStackVariables=TRUE  
- GenerateCFunctionCalls=TRUE  
These changes are, however, not as important as how effective the actual code is. A description of some common optimization techniques can be found in the appendix.
**Signal naming convention**

The LabVIEW C Code Generator generates C-style source code comments and variable names from the labels on the block diagram. Underscores ( _ ) replace special characters and spaces [27]. Generated variables must be unique. If multiple variables have identical names, the LabVIEW C Code Generator appends a sequential number to the end of the variable name to make the name unique.

The following table shows the LabVIEW signal source and corresponding generated C variable name and comment.

<table>
<thead>
<tr>
<th>LabVIEW Signal Source</th>
<th>Attribute Used</th>
<th>Block Diagram Example</th>
<th>Generated C Variable Name</th>
<th>Generated C Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front panel Control</td>
<td>Label</td>
<td>Motor On</td>
<td>Motor_On</td>
<td>/* Motor On */</td>
</tr>
<tr>
<td>Constant</td>
<td>Label</td>
<td>Trigger Level</td>
<td>Trigger_Level</td>
<td>/* Trigger Level */</td>
</tr>
<tr>
<td>Function</td>
<td>Label, Terminal Name</td>
<td>Over Voltage</td>
<td>Over_Voltage_x_y__</td>
<td>/* Over Voltage: x &gt; y*/</td>
</tr>
<tr>
<td>Loop tunnel</td>
<td>Source name, “LT”</td>
<td>For Loop</td>
<td>Trigger_Level_LT</td>
<td>/* Trigger Level: Loop Tunnel */</td>
</tr>
<tr>
<td>Case selector</td>
<td>Source name, “CS”</td>
<td>True</td>
<td>Relay_CS</td>
<td>/* Relay: Case Selector */</td>
</tr>
<tr>
<td>Case tunnel</td>
<td>Structure Label, “CT”</td>
<td>My_Structure</td>
<td>My_Structure_CT</td>
<td>/* My Structure: CT */</td>
</tr>
<tr>
<td>Shift register</td>
<td>Source name, “SR”</td>
<td>While Loop</td>
<td>Init_SR, Init_SR_1</td>
<td>/* Init: SR <em>/, /</em> Init: SR 1 */</td>
</tr>
</tbody>
</table>

13
2.1.2 C to Assembler

C compiles to assembler in three steps. The preprocessor, the “first and second pass” and finally the Linker. The C preprocessor is a separate program initialized by the compiler and its most common use is including other files, such as header files etc [15]. The preprocessor also remove comments, replace strings that have a defined value and remove unneeded characters [16]. The source file that was sent to the compiler consists now of only an ASCII text file with all the code from the original source code and all the code from the including files. This ASCII text will then pass thru something called the “first and second pass”, these stages of the compiler takes the ASCII text and parse it, checks the
syntax and transforms all the basic functions of the code into machine code [16]. The *.obj file that is created in the “first and second pass” is then sent to the Linker. The Linker is a program that takes the objects from the compiler and assembles them to a single executable program [10]. The Linker may also do an optimization operation to reduce execution time [16].

2.1.3 G to C to Assembler examples

Below are three examples how G can be generated to C. The first is an example of how a simple LV application is transformed into machine code. The first step when G is generated into C code is platform independent and can be viewed in figure 2.

![Figure 2. Transformation between G code to C](image)

The next step when compiling the C code to assembler is hardware dependent and is decided by the third party compiler used. If the graphical code above is to be used, for instance, on an Intel x86 CPU architecture the machine code could look something like this:

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>04BC7FFB</td>
<td>85C758CC 0000002D8 00000000 02E0B58B</td>
</tr>
<tr>
<td>04BCB00B</td>
<td>F0900000 000003D 04840F00 88000000</td>
</tr>
<tr>
<td>04BCB01B</td>
<td>89068B3E DC8D89C1 88000002 0002E0B5</td>
</tr>
<tr>
<td>04BCB02B</td>
<td>ECBD8D00 83000002 840000FE 0000001A</td>
</tr>
<tr>
<td>04BCB03B</td>
<td>8850368B 04478906 0847C758 00000004</td>
</tr>
<tr>
<td>04BCB04B</td>
<td>8904C683 0014E937 07C70000 00000004</td>
</tr>
<tr>
<td>04BCB05B</td>
<td>000447C7 C7000000 00408487 85800000</td>
</tr>
<tr>
<td>04BCB06B</td>
<td>000002EC 462806B8 8D058908 0002D4BD</td>
</tr>
<tr>
<td>04BCB07B</td>
<td>83078900 0002DCBD 8F000000 00000005</td>
</tr>
<tr>
<td>04BCB08B</td>
<td>00006DE9 EC858D00 88000002 08460306</td>
</tr>
<tr>
<td>04BCB09B</td>
<td>B08D0689 00000204 85880789 00000204</td>
</tr>
<tr>
<td>04BCB0AB</td>
<td>003D008B 0F000000 00000584 0011E900</td>
</tr>
<tr>
<td>04BCB0BB</td>
<td>85880000 0000020D 0000105D 00858900</td>
</tr>
<tr>
<td>04BCB0CB</td>
<td>88000002 7836845 840F0018 000000F1</td>
</tr>
<tr>
<td>04BCB0DB</td>
<td>02DC85BB 8D880000 000002D8 3901C183</td>
</tr>
<tr>
<td>04BCB0EB</td>
<td>088D0F01 89000000 0002D88D FF93E900</td>
</tr>
<tr>
<td>04BCB0FB</td>
<td>8D50FFFF 0002B885 5658CC00</td>
</tr>
</tbody>
</table>

As mentioned above the C code which is generated from the VIs that the programmer creates can be viewed in uVision and that is what the next two examples show. The first VI shows how notifiers share data between two loops and how semaphores in these loops are protecting critical code in the application. The dataflow shows us that on the right side the notifier functions is wired with a string constant called “Notifier Example” this is the name that the code will use when describing the reference to these particular notifiers. After the code has been generated the user can open uVision double click on the .c file which have the same name as the VI and then search for the variables in the code. Figure 3a and 3b shows the VI in question and the part of the C code where the notifiers have been described.
heap->dw_Constant = 0;

/*SetSignalReady( 0xA, 2);*/ /* dw_Constant */
SetSignalReady(106, 1);

MCB2300_GlobalConstantsHeapPtr->i0A44E850 = PDAStrNewFromBuf(_LVT("Notifier Example"),(uInt32)16);
heap->s_Constant_2 = MCB2300_GlobalConstantsHeapPtr->i0A44E850;

/*SetSignalReady( 0x24, 0);*/ /* s_Constant_2 */
SetSignalReady(212, 1);

MCB2300_GlobalConstantsHeapPtr->i0A44E7A8 = PDAStrNewFromBuf(_LVT("Notifier Example"),(uInt32)16);
heap->s_Constant_1 = MCB2300_GlobalConstantsHeapPtr->i0A44E7A8;

The other example is a little more complicated and uses functions that are being provided by NI (a tier 1 device), i.e. the low level coding to access hardware peripherals have already been made, to control a LCD screen. The code consists of a state-machine which controls the LCD display and sends information to the screen depending of what the user chooses. The state which is shown on the figure below shows the text that is displayed when an external button (INT0) is pressed. When looking in the created C code one can easily find all the variables, for instance the text “Welcome to NI” that is displayed on the LCD screen. In figure 4a and 4b the VI and the code can be viewed.
case 0 : {
    heap->dw_Increment_x_1 = heap->dw_Constant_CT;
    heap->l_y = 16;
    heap->dw_Y_1_2_ = 1;
    MCB2300_GlobalConstantsHeapPtr->i0A1B25E0 = PDAStrNewFromBuf(_LVT("Welcome to NI"), (uInt32)13);
    heap->s_String_in = MCB2300_GlobalConstantsHeapPtr->i0A1B25E0;
    PDAStrIncRefCnt(heap->s_String_in, (uInt16)1); /* BDConst - no arr/cls */
    nStep++;
}

Figure 4b. Part of the generated C code where string is being sent to the LCD screen.

A comment can be made about the two examples above. Readers that have been working on embedded systems can easily see the benefit of graphical programming in these examples. After merging the two programs, the built in functionality of LabVIEW lets the user just drag them into each other and its done, we have a program that have taken the author about two full days to produce. If this functionality would have been done by hand in for instance C it would have taken enormously more time. The use of the RTOS, RTX, makes the code even more efficient and if that would be implemented in C the development time would increase further. Actually, when this program is generated to C it consists of more the 75000 rows of code.

2.1.4 LabVIEW performance compared to C

There have only been limited compares between C and LabVIEW published from National Instruments. However a National Instruments LabVIEW product manager named Elijah Kerry has done some research in this issue. The study he performed is not typically comparing LabVIEW against the generated C-code but between LabVIEW programs against human made C-code. But because LabVIEW generated C-code use the same LUT and C-libraries as normal LabVIEW programs one can assume that the benefits from normal LabVIEW programs would apply to generated C-code as well.
The first test Kerry performed was to compare built in functions in LabVIEW against human created C-functions. These C-functions came from a open-source program so its easy to assume that companies who charge people for their C-programs could even be better, in either case the results can be viewed in figure 5

![Bar chart showing time comparison between C and LabVIEW](image)

**Figure 5: Time comparison between two LabVIEW versions and C**

The figure clearly shows that LabVIEW lack the speed as C-functions have in most cases but they are in the same time-domain and the minor differences is mainly because there is one additional step in the compilation process from LabVIEW to machine code compared with C-code. Also if the program is being runned on a Windows computer the lack of determinism can make these minor differences to be neglected.

A more recent study with both LabVIEW programs and C-functions show a similar result. Elijah Kerry this time made a project containing seven algorithms written in both C and LabVIEW. The C code had been compiled into an executable that is called and timed by a VI in the same way that a VI is timed in LabVIEW. The work of making a fair comparison was extensive. Below is figure 6 which is kind of hard to interpret but it shows a staple chart of the averaged time of between 100 to 10 000 iterations from various C and LabVIEW functions. Where the staples 0 and 1 shows Array manipulation, 2 and 3 FFT functions, 4 and 5 Fileread, 6 and 7 Filewrite, 8 and 9 Matrix algebra, 10 and 11 savage, 12 and 13 Sieve.
The conclusion from this is also that C programs and LabVIEW times usually run in the same division except some a little more advance functions in this particular case, however, one should be aware of that the C-calls in this test are very basic and some of them can be seen in the appendix.

2.2 LabVIEW Embedded

The burst of the 2001 It bubble NI put a lot of money in R&D to develop new products, they especially wanted to expand towards embedded development. The first step in this process was the LabVIEW embedded module that was added to LabVIEW 7.1 which came in 2004. This later changed name to NI LabVIEW Microprocessor SDK (Software Development Kit). LabVIEW SDK where specifically developed for applications incorporating 32-bit embedded microprocessors like PowerPC, ARM, TI C6x, x86 and Coldfire [22], this new development module incorporated approximately 400 analysis functions for signal processing, linear algebra, curve-fitting, statistics and calculus. The development of LabVIEW Embedded and LabVIEW SDK allowed hardware developers with little or no software experience to create their own code by clicking on the block diagrams of a software interface [23]. This new graphical way of programming embedded
applications were a great tool for both “Domain Experts”, people with great knowledge in a specific area but limited programming skills, and “Embedded Experts”, people that needed an easier and faster development process. The strength of LV SDK was also one of its weaknesses; the broad coverage of different microprocessors makes LV SDK expensive and initially hard when a new microprocessor is ported.

2.2.1 LabVIEW Embedded Module for ARM
LabVIEW Embedded Module for ARM was developed after the LV SDK to specifically deliver graphical programming to the popular embedded 32-bit reduced instruction set computing (RISC) processor. This module provided a comprehensive development environment for ARM7, ARM9, and Cortex-M3 microcontrollers. The decision to do this version of the SDK is that the ARM family accounts for approximately 75% of all embedded 32-bit RISC CPUs, making it one of the most widely used 32-bit architectures [24]. The benefit of having software that specifically targets ARM architecture is that it’s easier to start and the cost is lower. Other than the obvious limitation towards Microprocessor SDK the two programs are very similar.

2.3 HM20 S-2 Module
The HM20 S-2 module houses both an AT91RM9200 ARM processor and a Spartan 3E FPGA. The FPGA is directly connected to the boards I/O and is a true real-time data processor that may serve many purposes: DSP, flexible display interface, high-speed parallel I/O control, custom interface gateway etc. [18]. The processor runs the Windows Embedded CE or Linux operating systems for easy accessing and usage. The starter kit is very suitable for fast prototyping, and a fast way to production as software development can proceed during the development of a customized I/O board. The HM20 S-2 can easily be mounted with a LCD touch panel which can be controlled by the ARM processor.

2.3.1 ATMEL AT91RM9200
The AT91RM9200 is a microcontroller with ARM9 architecture. It is based on the ARM920T 200-MIPS advanced RISC processor. The ARM920T technology is designed for use on a wide range of platform OS and to be used in next generation smart phones, 3G baseband and applications processor, audio and video decoding etc [18]. The AT91RM9200 microcontroller offers a flexible configuration of on- and off-chip memories together with an extensive set of peripherals for control [17]. It houses USB Full Speed, 10/100 Base T Ethernet, MCI, three SSC, four USART’s, Master/Slave SPI, Timer Counters and Two Wire interface [17]. Also the AT91RM9200 has a fully featured system controller for efficient system management, including two on-chip oscillators and PLL’s, clock generator, AIC, DBGU, periodic interval timer, watchdog timer and real-time clock with alarm interrupt [17].

2.3.2 FPGA Spartan 3E
The FPGA Spartan 3E is Xilinx seventh family in the low cost Spartan-series and the third Xilinx family manufactured with 90nm process technology [19]. The Spartan 3E series consists of five devices with features ranging from 100K to 1.6M system gates, 66 to 376 I/Os, up to 648 Kbits of block RAM and 231 Kbits of distributed RAM, up to 36 embedded 18x18 multipliers and up to eight Digital Clock Managers [19, 20].

2.4 LabVIEW training
Before the actual work on this thesis begun, the LabVIEW syntax had to be studied. This was made thru various National Instrument courses. The ones that felt the most essential were:

- Basic I
- Basic II
- Intermediate I
- Intermediate II
- CompactRIO
- Data Acquisition and Signal Conditioning (DAQ)
The LabVIEW Basic courses teach how to create multiple loop applications and learn about the communication techniques needed to share data between the loops. After the courses you are able to design and create stand alone applications. The LabVIEW intermediate courses is an extension of the Basic courses and teaches how to design and create large applications that are flexible, scalable, and maintainable with communication technologies including DLLs, TCP, UDP, and .NET.

The DAQ course implements real life measurements and hardware monitoring, analog input, analog triggering, signal conditioning, signal processing, analog output, digital I/O, and counters were also covered in this course. CompactRIO is National Instruments hardware consisting of a FPGA processor for deterministic I/O and control functions and a real time controller for processing and communication. This course was a good introduction to embedded programming.

2.5 LabVIEW embedded for ARM training on evaluation boards

The LabVIEW embedded for ARM software supports two development modules out of the box. These are the EK-LM3S8962 and the MCB2300. When developing graphical code to any of these modules one only have to select a target in the getting started menu and then choose either the MCB2300 or the EK-LM3S8962 in the Select ARM target type menu, see figure 7. LV ARM, in addition with basic programming and mathematics, then supply the user with palettes for controlling of elemental I/O, interrupts, board peripherals like led’s and buttons. It also has controls for displays and SPI and I2C communication. The steps for compiling and flashing the boards are also fully supported and no significant settings have to be made manually.

![Figure 7: The two steps when using a supported development board.](image)

Tier 1 and Tier 2 devices

There are two tiers of support for ARM Microcontrollers in LabVIEW. The two development boards mentioned above are validated Tier 1 devices that provide an out-of-the-box programming experience. In addition to basic programming and mathematics, tier 1 devices also provide LabVIEW palettes for communications, analog, and digital I/O. Tier 2 devices provide processor support, but no specific palettes are provided, so this requires you to develop and integrate drivers. Figure 8 shows the feature breakdown of tier 1 and tier 2 processors.
2.5.1 EK-LM3S8962

The Stellaris LM3S8962 Evaluation Kit is a compact and versatile evaluation platform for Ethernet + CAN, the CAN module is on a separate board. Each board has an In-Circuit Debug Interface (ICDI) that provides hardware debugging functionality not only for the on-board Stellaris devices, but also for any Stellaris microcontroller-based target board [33]. The evaluation kit offers analog input, a speaker, OLED display, Ethernet, and eight LEDs and can be viewed in figure 9. Because the EK-LM3S8962 is a tier 1 device it comes with a variety of pre built functions, these functions gives the user access to hardware peripherals, on a tier 2 device the user would have to create a framework to get the same functionality, frameworks is described in section 4.1. LabVIEW for ARM supports these Vis as a default:

- Display Draw Circle
- Display Draw Line
- Display Draw Pixel
- Display Draw Rect
- Display Draw String
- Display Get String Width
- Display Init
- Display Set Background
- Display Set Clip Region
- Display Set Foreground
- Display Update

<table>
<thead>
<tr>
<th>Basic Programming Structures</th>
<th>✔</th>
<th>✔</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric, Boolean Logic</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Arrays, Clusters, Matrices</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Semaphores, Queues, FIFOs</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Mathematics, Signal Processing</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Statistics</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>TCP/IP, Serial Communication</td>
<td>✔</td>
<td>×</td>
</tr>
<tr>
<td>Analog I/O</td>
<td>✔</td>
<td>×</td>
</tr>
<tr>
<td>Digital I/O</td>
<td>✔</td>
<td>×</td>
</tr>
<tr>
<td>PWM</td>
<td>✔</td>
<td>×</td>
</tr>
<tr>
<td>I2C, SPI Communication</td>
<td>✔</td>
<td>×</td>
</tr>
<tr>
<td>LCD</td>
<td>✔</td>
<td>×</td>
</tr>
<tr>
<td>Cycle-Accurate Simulation</td>
<td>✔</td>
<td>?</td>
</tr>
</tbody>
</table>

Figure 8: Table of the differences between Tier1 and Tier2 devices
2.5.2 MCB2300
The Keil MCB 2300 introduces the programmer to the NXP LPC2300 ARM family from Philips. The kit is similar to the LM3S8962 board but with some limitations. The processor is not quite as powerful, there is no CAN module included in the development kit and the OLED screen is replaced by a little more simple screen. The LabVIEW functionality is however the same regarding pre-built functions and out-of-the-box simplicity. In Figure 10 the MCB 2300 board can be viewed.

3: Testing
3.1 Porting LabVIEW ARM to a processor
When using any other ARM processor than those mentioned above a porting process has to be followed. NI has two documents describing the basics of this process [29] and [30]. A full explanation of all the steps necessary can be found in the appendix.

A general process for incorporating a new ARM target follow these general steps:
• Porting the RTX Real-Time Kernel
• Integrating the Real-Time Agent module for debugging
• Developing peripheral and I/O drivers (elemental I/O).
• Creating the target in LabVIEW and incorporating the Keil toolchain

RTX Real-Time Kernel is a RTOS for ARM7, ARM9 and Cortex-M3 devices. As any RTOS it allows the user to create programs that simultaneously perform multiple functions or tasks. The biggest benefit of using an RTOS is the numerous scheduling, maintenance and timing issues that the user doesn’t have to be concerned with. Another issue that is beneficial in this particular case is that LabVIEW for ARMs default setting is to compile its G-code into RTX supported C-code which means that less settings has to be made if the target supports RTX. The reason for RTX is that it is a widespread and acknowledged RTOS and also because Keil, the maker of the C to assembler compiler, is the developer of RTX.

Real-Time Agent is a small code module that can be added to ARM applications. It allows the program to communicate back to the µVision Debugger using the ULINK2 USB-JTAG Adapter. The Real-Time Agent offers some enhanced debugging capabilities including:
• Memory Read/Write During Program Execution
• Terminal Emulation
• Serial Debug Output

The benefit of using the Real-Time Agent for debugging is that the programmer can test various applications before implementing the hardware. For instance one can simulate external hardware buttons by just adding software buttons on the LV front panel, then after the functionality have been verified the software buttons can be replaced by hardware.

Developing peripherals and I/O drivers is a one time, time consuming and difficult task, it includes creating elemental I/O and changing settings in uVision so the hardware can then be accessed in an appropriate way. However, this and all the other steps in this part can be neglected if a Tier 1 device is used.

Creating a target in LabVIEW refers to the creation of an embedded project which includes all the necessary plug-ins and elemental I/Os in addition with hardware specific information provided from uVision.

3.2 Flashing the ARM processor

The actual flashing of the processor on a proper set up machine is done automatically by LV ARM i.e. the user just press the run button in LV in the same manner as the user would have done when running the program in Windows and the program will be compiled and flashed to the target. When using a supported Tier 1 device a USB to JTAG-ICE converter takes the bit-stream from uVision and streams it to the processor. When porting to a Tier 2 processor the user have the choice to either use JTAG-ICE or thru TCP/IP. When porting a Tier 2 device, additional difficulties such as memory placement and boot-loaders have to be tackled but are described in the appendix. The HM20 platform unfortunately doesn’t have a JTAG-ICE connector, the AT91RM9200 have a JTAG-ICE interface so it’s possible to modify the HM20 to add this functionality of the board. The HM20 has, however, a TCP/IP interface and a Linux core that can be used to transfer code thru FTP. A step in trying to get the compiled code from LV ARM to the HM20 was true the Linux interface. To accomplish this, actual C-code generated from LabVIEW is needed to be used. Easy enough, in uVision all the generated and depending files can be viewed and the actual program is found in a .c file with the corresponding name as the VI. When all the code have been located there are essentially two ways of getting it to work on the HM20s Linux environment. One is to use the Batch file which uVision creates and move it thru a FTP server onto the HM20 and then use the bit-stream from that to flash the microprocessor. Unfortunately this process failed because of Windows – Linux differences and the Batch file could not be opened. The second method is to move all the generated C code to the Linux platform and then compile it and flash it there, due to the fact that the generated C code is specified to be run with uVision the author could not get this method to work. However, it’s very likely that a person with better knowledge on Linux could get this compilation to succeed.
3.3 FPGA programming with LV
The LabVIEW FPGA module that National Instruments provide can compile the G code to VHDL code in the same way as LabVIEW for ARM can compile G to C. However the difference is that LV FPGA can only be used with National Instruments own products like the CompactRIO, which houses a Xilinx FPGA, NI R-series DAQ devices and FlexRIO. These limitations will make it impossible to program the FPGA that is housed on the HM20 using LabVIEW. With this said there is no guarantee that this will always be the issue, there could be a time when the VHDL compiler will be opened to third party FPGAs in the same way as the LV ARM code.

3.4 LCD testing
Due to the lack of JTAG-ICE interface and problems of getting around this issue the LCD on the HM20 will unfortunately not be able to be tested within the scope of this thesis.

4: Implementation
4.1 Creating framework
Framework is a term describing the part of the program which is hardware specific and includes the creation of elemental I/Os. These elemental I/Os include the access and control of hardware peripherals, hardware timing and accessing port and pins etc. This is also the only part when using LV ARM that the programmer has to have deep knowledge about embedded programming and hardware specifications; luckily this step only has to be made once.

When creating a framework the programmer should first decide what peripherals or ports that should be accessed, a decision not to access the many ports that microprocessors have will save the project considerable amounts of time. For instance if the programmer only needs to access port A and B there is no point creating a framework which includes port C and D as well.

The link between the hardware specific peripherals and the graphical code G lies true a function block called Inline C-node which is only available in LabVIEW if you have LV module for ARM or SDK installed. The Inline C-node is a frame where the user can draw wires in and out to send and receive data; the user can then combine this data with written C-code to access registers and ports on the microcontroller target. The Inline C-node even have the ability to import header files which means that premade functions and third-party drivers can be used in LabVIEW using simple function calls. The normal procedure when creating a framework for accessing I/O peripherals would be to open an empty VI, putting an Inline C-node in the middle, drawing wires that represent the data and settings for the microcontroller, writing code in C that transforms this LabVIEW data into information that the microprocessor understands and then save the VI as a elemental I/O. After these steps have been completed and regarding how general the programmer has made his VI it can be dragged and dropped in various embedded Vis. The most interesting part when using the inline C node in pre-made VIs like this is that users with no knowledge about embedded C-programming can in a easy way access deep hardware specific ports, peripherals and drivers without writing a single line of code. The recommendation would therefore be that a company hires one or more persons to create a general framework of elemental I/Os that users with limited knowledge in the company later can use.

Another benefit of using modular VIs like the one explained above is that it can be altered without changing the rest of the program. For instance, imagine that there is a VI in a program (subVI) which is controlling data communication, perhaps a CAN protocol. This VI might have two inputs and two outputs that the main program uses, sending and receiving data and who to send to and from which target was the data being sent. If the company later decides that the communication protocol are going to be replaced to for instance SPI, only the VI controlling the communication needs to be changed and the rest of the program can be kept untouched, but only if the new subVI has the same amount of inputs and outputs of course. This functionality is not reserved for embedded applications but is indeed a feature of all LabVIEW projects.
4.1.1 Creating Custom Elemental I/O for ARM

Elemental I/O is a flexible, user-defined way to access analog and digital I/O peripherals from LabVIEW on embedded ARM targets. Elemental I/O nodes are portable across many targets, so even if the underlying implementation of the driver changes among targets, you do not need to modify the LabVIEW application when migrating code to another processor [29]. An I/O device is the software representation of hardware that performs I/O on an ARM target. It is a collection of I/O resources, pins, and plug-in VIs that define the elemental I/O for a target. You can reuse an I/O device on multiple targets so several similar targets can share a single I/O device [29].

Elemental I/O classes
Elemental I/O supports the following classes:

- Analog In
- Analog out
- Digital In
- Digital Bank In
- Digital Bank Out
- Pulse Width Modulation Out

Pins
A pin is the software equivalent of a physical pin that connects an I/O device to the outside world. Resources can share pins because, typically, there are fewer pins than resources. Pins are optional. Use pins if you have conflicting devices [29].

Elemental I/O Node
In LabVIEW, users place elemental I/O nodes on the block diagram by dragging the node from the project palette onto the block diagram. When the user creates an elemental I/O device, he defines which of the I/O classes that can be used. Figure 11 shows a project that supports four elemental I/O classes. When the Elemental I/O Node executes on the embedded target, it executes LabVIEW code that the elemental I/O device defines. The elemental I/O device has plug-in VIs for each type of I/O the device supports. These VIs replace the Elemental I/O Node when executing on the embedded target. Most of the plug-in VIs has a pin number as an input, and compute the I/O result using an Inline C Node that accesses the I/O registers [29].

![Figure 11: View of project explorer.](image-url)
Creating elemental IO is a two part process, the first step involves programming plug-in VIs using inline C node and generic input/outputs to set all the correct registers etc. The second step revolves around getting these VIs into the LV environment using the Elemental IO Device Wizard, a detailed explanation of this can be found below and in the appendix.

Writing Elemental I/O Plug-In VIs for the AT91RM9200
Elemental I/O is a flexible, user-defined way to access analog and digital I/O peripherals from LabVIEW on embedded ARM targets. Elemental I/O nodes are portable across many targets, so even if the underlying implementation of the driver changes among targets, you do not need to modify the LabVIEW application when migrating code to another processor. Therefore, use elemental I/O to create block diagrams that you can reuse on many platforms [29].

The first step in the process of writing Elemental I/O is creating the plug-in VIs. If you try to run the Elemental I/O Wizard before creating the plug-in VIs an error will be received when eio.xml file is created, which is the file that defines how the element I/O is implemented for the target. If the plug-in Vis is modified in any way the wizard must be ran again and LabVIEW have to be restarted to ensure that the information in the target matches the Vis [29]. LabVIEW for ARM have template Vis which include all possible controls and indicators for the plug-in Vis. Copy the template Vis for the target, replace the Project Attributes and Node Attributes cluster controls with type definitions or strict type definitions and remove controls that you don’t need.

The steps required for implementing elemental I/O can be found in the appendix

5: Analysis of technology

5.1 Possible markets
Due to the very large spread of ARM architecture the various markets of LV for ARM is almost endless. Especially in companies where there are expertise in other areas rather than actual embedded programming but still have the desire to distribute applications to customers. The biggest advantage lies for existing LV users which are working with hardware implementation. The benefit for these customers is that they can use existing knowledge and applications to embedded products; i.e. a user can easily develop and test an application with LV on the NI cRIO platform, programmable automation controller which houses a FPGA and power PC with capabilities to include hundreds of modules. Then when the application works on this a bit more expensive, but easier to use, hardware the tested application can be distributed on hardware using an ARM architecture. Then for all the people who have an idea of an embedded application they want to market but they lack in programming knowledge LV ARM is perfect. Of course the initial cost will be higher with LV ARM but with an easier learning curve and stable and instant code the price will most likely be less.

5.2 Limitations
The biggest limitation of LV ARM is of course that it only can be used on processors with ARM architecture, the workaround for this is to use Microprocessor SDK but the limitation still lies in hardware because, as mentioned, SDK can only be used with 32 bit processors. When using Tier 1 devices the only limitation is that LabVIEW knowledge is needed, however the learning curve for LV lies in the time span of weeks compared to years when talking about embedded programming with C or assembler. When using Tier 2 devices the challenge of creating a framework is the biggest obstacle, this can be solved by hiring a consultant or letting a few more experienced co-workers define and create necessary elemental I/O for users with limited embedded knowledge.

Other limitations lie in documentation and the question marks of how effective the G to C generator really is to cope with time constraints and memory usage. However, the author believes that this limitation only affects companies which are in the medical or equivalent business. Due to its very high demand of documentation, certification and safety these companies only use languages that have been in use for several years and the change of these is a very slow process. For instance pacemakers are almost exclusively being developed using Assembler, which means that going any higher in the
The chosen language will be rejected due to safety issues, however the lack of ARM and 32-bit processors in pacemakers makes LV for ARM unaffected by this limitation.

It's however important to state that a programming language can’t be certified, its application can but the actual programming type cannot. This is why the use of LV for ARM in medical, aeronautical or military applications is far from impossible because like all other programming languages it can be put thru a certification process and if it passes all the requirements there is nothing that points that LV ARM would be more affected by bugs or software failures, on the contrary the easiness of the LV graphical implementation improves the rate of finding and removing bugs during development and testing.

6: Results

The process of porting the ARM processor was smooth up until the actual programming of the HM20, the lack of JTAC-ICE connection on the board made it impossible to get the actual bit-stream to the AT91RM9200 mounted on the board. The techniques of manually trying to transfer a simulated batchfile and trying to transfer the code via TCP/IP was unsuccessful. Due to time limitations and the unwillingness from the author's part so solder on the HM20 card the implementation of a JTAG-ICE interface will not be done.

The use of tier1 devices and the LabVIEW for ARM tool chain was however a great success. The author was surprised in how easy it was to start doing complicated embedded applications with powerful hardware just out of the box. The literature study also showed that the generated code is not as bad or hard to read as many would perhaps believe. The surprise the author felt for the easiness of using Tier 1 devices was shared with a lot of people when LV ARM applications was showed on the 2008 embedded fair on Ålsvöömmässan in Stockholm.

7: Discussion

Programmers with a background in conventional programming often show a certain reluctance to adopt the LabVIEW dataflow scheme, claiming that LabVIEW is prone to race conditions. Race conditions arise in software when separate processes or threads of execution depend on some shared state [38] i.e. a function that can be accessed by two or more processes but have the criteria that they should do so in a specific order to prevent failure will increase the chance of race conditions if there is no synchronization or control of the function calls in the application. In reality, this stems from a misunderstanding of the data-flow paradigm. The above mentioned data-flow (which can be forced, typically by linking inputs and outputs of nodes, see the What is Graphical programming section) completely defines the execution sequence, and that can be fully controlled by the programmer. Thus, the execution sequence of the LabVIEW graphical syntax is as well-defined as with any textually coded language such as C, Visual BASIC, and Python etc. Furthermore, LabVIEW does not require type definition of the variables; the wire type is defined by the data-supplying node. LabVIEW supports polymorphism in that wires automatically adjust to various types of data, this feature is however not recommended to use but does make it easier for beginners to make powerful applications. Therefore temporary variables with useless names have been supplanted by the anonymity of wires, which show the data's purpose and destination far better than names like “temp,” “foo,” and “dummy” [23].

8: Conclusion

The LabVIEW programming environment, with the included examples and the documentation, makes it fast and simple to create small applications. This is a benefit on one side, but there is also a certain danger of underestimating the expertise needed for good quality "G" programming. For complex algorithms or large-scale code, it is important that the programmer possesses an extensive knowledge of the special LabVIEW syntax and the topology of its memory management. The most advanced LabVIEW development systems offer not just applications using embedded code but with .NET, databases, VHDL, Instrumental control and Matlab scripts. This in addition with the incorporation of a module such as Vision, Machine control, Signal express or Diadem makes LV a language that is diverse and powerful. Furthermore, it is possible to create distributed applications, such as exe files and installers, which communicate by a client/server scheme, and are therefore easier to implement due to the inherently parallel nature of G-code.
9: Recommendation

The general recommendation for a user concerning LV ARM is that he/she should make a calculation where the time benefits with LV ARM is compared with its higher price. This calculation should incorporate cost and time for initial porting and elemental I/O creation, learning curves for both experienced users and new employees, development time, time for bug searches and cost for maintaining the code. If the cost is lower for LV ARM well then the choice is obvious and if the price for using LV for ARM is similar or a bit higher I would still recommend LV ARM because when companies determine development time and bug fixes they usually pick a to low value.

The recommendation for Prevas is to incorporate a JTAG/Ice connector on their HM20 platform; this will make it very easy to use LV for ARM and even other third party compilers.

National Instrument should increase their variety of Tier 1 devices, both with simpler and more advance versions. This is because the easiness of using a Tier1 device compared to a Tier2 device is vast. Perhaps NI could partner up with a hardware company where customers can customize the hardware and NI provides the completed framework. Providing framework is also a very good idea for other companies which they might be able to sell. This is mainly to increase the selection for customers so that they can purchase an ARM processor with the exact functionality that they need for their application. Also with a wider selection of Tier 1 devices and pre built framework for hardware peripherals users would show less reluctance to move to a new development platform.

10: References

Books


Articles


Webpages

11: Appendix
Porting LabVIEW ARM to a processor walkthrough

The steps in the specific tutorials provided by NI describe how to port a Philips NXP LPC3180 ARM9 microcontroller. To incorporate an arbitrary ARM microcontroller, you need to custom tailor many of the step details provided in the tutorials to the specific microcontroller. These steps will be described in detail with the AT91RM9200 microcontroller as model below.

Integration of LabVIEW and Keil toolchains
The first step in porting a custom ARM processor is to install the Keil uVision3 software that is provided with the LabVIEW for ARM software or you can download it on www.keil.com. When installing uVision3 you can choose on which target to use, because I have no plan of doing the actual flashing with uVision3 I chose simulated hardware so that access to the generated C code is gained. When uVision3 is installed the LabVIEW for ARM module can be installed. After the software is installed the actual porting can begin. To port an ARM target to LabVIEW, the target must support the RTX Real-Time Kernel [29]. So the first step in the porting phase is to check if your microprocessor already supports RTX Real-Time kernel, one can do that by browsing the Keil Startup directory and look for a RTX-Conf*.c file for your target, luckily the AT91RM9200 have such a file. If a C file for your microprocessor don’t exists the given tutorial [29] have information for porting RTX to your ARM microcontroller. LabVIEW for ARM also supports live front panel debugging, but if one wants to use that feature the target must also work with the Real-Time Agent module. The AT91RM9200 is not supported in advance but the tutorial shows how to add the module to your application. Because the HM20 board does not have a front panel this feature will not be used in this project.

Use the Target Editor to Manage Embedded Targets
All of the ARM targets for LabVIEW use the same plug-in VIs. Therefore, to ease the process of adding a new ARM target to LabVIEW, a template target has already been provided that contains a TgtSupp.xml file configured with the correct plug-in VIs, folder structure, and placeholder driver files. Creating a new target for LabVIEW involves making a copy of this template target and giving it a proper name that clearly identifies what the target is.

- Browse to the LabVIEW\Targets\Keil\Embedded\RealView directory.
• Make a copy of the Generic folder and rename it AT91RM9200 (Or the name of the microcontroller that will be ported).
• Browse to the LabVIEW\Targets\Keil\Embedded\RealView\AT91RM9200\LPC2378 directory and copy everything in this folder.
• Browse to the LabVIEW\Targets\Keil\Embedded\RealView\AT91RM9200\ directory and paste everything that was copied from the LPC2378 folder.
• Delete the LM3Sxxxx and LPC2378 folder.
• Open LabVIEW and select Tools»Embedded»Target Editor.
• Select File»Open, then browse to the AT91RM9200 folder renamed in the previous step.

Select File»Open, then browse to the AT91RM9200 folder renamed in the previous step

Locate and open the TgtSupp.xml file.

Right-click the Other icon and rename it AT91RM9200.

Right-click Applications and select Properties
On the Category page, select Manage Interrupts and delete this entry by clicking the Red X Remove Property Page Button.

- Click OK to return to the Target Editor.
- Select File→Save, then exit the Target Editor. You have now created the target.
- Close LabVIEW completely and open LabVIEW again so that the available targets are reenumerated.

**Prepare the Template files**

LabVIEW relies on a properly configured µVision project that defines where to locate various driver files that are target-specific. Because these driver files are target-specific, some of the driver files included in the generic templates are reusable while others are not. The steps to prepare the appropriate driver files for the AT91RM9200 are described in the tutorial.

- Browse to the LabVIEW\Targets\Keil\Embedded\RealView\AT91RM9200\eio directory and delete everything in this folder. You will create elemental I/O drivers for the target in a later chapter.
- Browse to the LabVIEW\Targets\Keil\Embedded\RealView\AT91RM9200\interrupt\ directory. Delete all files in this directory except the interrupt.xml file. The interrupt.xml file for the LPC2378 configures how LabVIEW generates code when using interrupts. National Instruments recommend that interrupts are to be implemented after the initial porting has been made and are working correctly. That step will not be touched in this thesis, however for the user that wishes to use interrupt handlers a complete description can be found in this tutorial: [http://zone.ni.com/devzone/cda/tut/p/id/7063](http://zone.ni.com/devzone/cda/tut/p/id/7063) or by just searching on the National Instruments webpage.
- Open interrupt.xml in a text editor.
  - Delete all of the text in this file and replace it with the following:
    - <Version>1</Version>
    - <Array>
      - <Name>ProvidedInterruptData</Name>
      - <Dimsize>0</Dimsize>
    - </Array>
  - Save interrupt.xml and exit the text editor.
- In this step the RTX OS files for the target will be configured. Browse to the LabVIEW\Targets\Keil\Embedded\RealView\AT91RM9200\Template\System directory, and delete the RTX_Config.c and LPC2300.s files. These files will be replaced with microcontroller specific RTX files. Luckily the AT91RM9200 is already supported for RTX so RTX_Conf_RM9200.c and RM9200.s could be found in this directory C:\Keil\ARM\Startup\Atmel
- Open RTX_Conf_RM9200.c in a text editor.
  - Add the following header file:
    - #include <LVConfig.h>     /* LV Config header */
  - Close and save RTX_Conf_RM9200.c.
• Open RM9200.s in a text editor.
  o Comment out the following lines from the startup file:
    ▪ Undef_Handler B  Undef_Handler
    ▪ PAbt_Handler   B  PAbt_Handler
  o Add the following line to the startup file:
    ▪ IMPORT  Undef_Handler
    ▪ IMPORT  PAbt_Handler
  o Close and save RM9200.s.
  o These changes allow LabVIEW to set the correct error bits when an Undefined IRQ handler or Prefetch Abort handler fires.
• Browse to the LabVIEW\Targets\Keil\Embedded\RealView\AT91RM9200\Template\Project directory. Copy the Ext_RAM.ini, Int_RAM.ini and Retarget.c file found in the following directory for the AT91RM9200, C:\Keil\ARM\Boards\Atmel\AT91RM9200-EK\Blinky.
• Open Ext_RAM.ini in a text editor and modify the line that states
  o LOAD Ext_SDRAM\Blinky.axf INCREMENTAL
  o to the following:
  o LOAD Obj\LabVIEW.axf INCREMENTAL
  o Close and save Ext_RAM.ini.
  o The template µVision project is configured by default to compile the application into an executable named LabVIEW.axf in the Obj directory; therefore, this change ensures that the correct file is loaded into memory upon execution.
• Browse to the LabVIEW\Targets\Keil\Embedded\RealView\AT91RM9200\Template\Drivers directory. Delete the Ethernet, Interrupt, I2C, and SPI folders.
• Browse to the LabVIEW\Targets\Keil\Embedded\RealView\AT91RM9200\Template\Drivers\RTClock directory. Open ARM_RTClock.c in a text and delete the code for both functions, ARM_rtc_gettime() and ARM_init_rtc(). Note that only the code in the functions is to be removed, the functions should still be there so no linking errors will come during the compile process.
• Save ARM_RTClock.c and exit the text editor.
• Browse to the LabVIEW\Targets\Keil\Embedded\RealView\AT91RM9200 directory. Open TargetConfig.ini in a text editor.
• Delete all text below the line that states “[Files].”
• In the [Define] Section, add the following line: SocketSupport = 0

**Modify the Template µVision project**

When LabVIEW compiles a VI, it makes calls to µVision, directing it to open up a preconfigured project that has been customized to the specific target. This preconfigured project defines how µVision compiles the code. It also defines how the code is downloaded and run on the target [29]. Of course the necessary steps of how to do this is described in the tutorial. However, because µVision will be used only as a simulator the tutorial will not be followed to 100%. Most of these changes will be in the “Options for target” menu in µVision. In the Output tab check the “Create Batch File” to make a (*.BAT) file that contains all commands of the build process and in the Debug tab check the “Use Simulator” so you can test and debug your embedded application before the hardware is ready. µVision3 simulates a wide variety of peripherals including the serial port, external I/O, and timers [31]. These are the necessary steps to modify the µVision project.

• Open the µVision 3 environment.
• Open the Project menu and select Open Project....
• Browse to the LabVIEW\Targets\Keil\Embedded\RealView\AT91RM9200\Template\Project directory and open the LabVIEW.uV2 project.
• In the Project Workspace, under the Target Startup folder, remove the LPC2300.s file by right-clicking the individual file and selecting Remove File.... Also, remove the LPC23_EMAC.c, ARM_irq.c, ARM_I2C.c, and ARM_SPI.c files from the Target Drivers folder.
• Right-click the Target Startup folder and select Add Files to Group “Target Startup.” Browse to the LabVIEW\Targets\Keil\Embedded\RealView\AT91RM9200\Template\System and add the RM9200.s and Retarget.c files that were copied to this directory in the previous section.
• Configure the project options by selecting the Options for Target icon on the build toolbar.

On the Device tab, select the ATME<->AT91RM9200 in the CPU vendor database list.

Configure the Target settings as shown. Make sure the Operating system is set to use the RTX Kernel. Set the Code Generation for Thumb-Mode, make sure Cross-Module Optimization is checked, and make sure Use MicroLIB is NOT checked. The rest of the setting should be checked by μVision when the user chooses a microcontroller in the Device tab. If the Memory Areas are not configured refer to the manual of the microcontroller.

On the Output and Listing tab, leave the settings as they are. The name of the executable MUST be LabVIEW. Change to the Asm tab. In the Conditional assembly control Symbols, Define field, type RAM_INTVEC, REMAP. When set, the startup code copies exception vectors from on-chip flash memory to on-chip RAM, and from on-chip RAM to address 0. For documentation on the various SET symbols, see the top of the device’s startup file (RM9200.s).

In the Debug tab, select the Use: ULINK ARM Debugger option. The application for this thesis will not be used to flash the microcontroller directly; therefore the Simulator tab is checked instead.
Configure the ARM Target Driver Setup by clicking the Settings button. The ARM Target Driver Setup settings should be as follows with Max JTAG Clock set to RTCK and all Debug options checked (Cache Options, Download Options, and Misc Options).

Click OK to return to the Options for Target window. Be sure that the Load Application at Startup, under the ULINK ARM Debugger setting, is checked. Add the correct Initialization File by clicking the browse button and select the Ext_RAM.ini file that you previously copied into the same folder as the project.

In the Utilities tab, configure the Target Driver for Flash Programming by clicking the Settings button. Select the LPC2000 IAP2 S12kB Flash and click the Remove button. Click the Add button. In the new window, highlight the LPC3180 NAND Flash SP and then click Add to return to the previous window. Configure the remaining Flash Download Setup options as follows and click OK. Be sure to set the RAM for Algorithm Start and Size values correctly.

Back on the Utilities tab, add the correct initialization file by clicking the browse button and select the int_RAM.ini file that you previously copied into the same folder as the project.
Add support for the Advanced Analysis Libraries

- Browse to the `LabVIEW\Targets\Keil\Embedded\RealView\MC82300` directory. Copy the build directory, and paste it into the `LabVIEW\Targets\Keil\Embedded\RealView\AT91RM9200` directory. Under the AT91RM9200 directory, create a new folder named "lib."
- Browse to the `LabVIEW\Targets\Keil\Embedded\RealView\AT91RM9200\build` directory and open LVAnalysis.Uv2 in the μVision environment.
- Configure the project options by selecting the Options for Target icon on the build toolbar.

On the Device tab, select the ATML AT91RM9200 in the CPU vendor database list.

- In the Output tab, click on the Select Folder for Objects button and browse to the `LabVIEW\Targets\Keil\Embedded\RealView\AT91RM9200\build` directory and click OK.
- In the Listing tab, click on the Select Folder for Listings button and browse to the `LabVIEW\Targets\Keil\Embedded\RealView\AT91RM9200\build` directory and click OK.
- Click OK to return to the project workspace.
- Build the library by selecting either the Build Target icon on the build toolbar or the Project»Build target on the main menu bar.

If μVision gives the error that some files are missing the solution is to search for these files manually and then add the paths in the Include Path field under the C/C++ tab under Options for target.

- When the build is complete, an LVAnalysis.lib file will appear in the `LabVIEW\Targets\Keil\Embedded\RealView\AT91RM9200\lib` directory.
- Browse to the `LabVIEW\Targets\Keil\Embedded\RealView\AT91RM9200\` directory and open TargetConfig.ini in a text editor.
- Under the line that states [FILES], add the following:
  - `<LABVIEW>\Targets\Keil\Embedded\RealView\AT91RM9200\lib\LVAnalysis.lib = lib`
- Save and close TargetConfig.ini.
Create Elemental I/O
You implement elemental I/O by providing a set of configuration files, plug-in VIs, and XML files. Use the Elemental I/O Device Editor to define the I/O classes, pins, and resources for a device. The Elemental I/O Device Editor also takes the configuration files and plug-in VIs and creates the necessary files for implementing elemental I/O.

Prepare Template Files

- Browse to the LabVIEW\Targets\Keil\Embedded\RealView\AR91Rm9200\eio directory.
- Create four new folders in this directory:
  - EIO
  - AI
  - GPIO
  - PWM
- Browse to the LabVIEW\examples\lvemb\EIO\EIO User Plugin Templates\ directory.
- Copy the following files to the LabVIEW\Targets\Keil\Embedded\RealView\AT91Rm9200\EIO directory. They will be used as a template for creating the various plug-in VIs.
  - NodeAttr.ctl
  - PrjAttr.ctl
  - NodeAttrValidator.vi
  - PrjAttrValidator.vi
  - EIOImplementation – Read Scalar.vi
  - EIO Specify Node.vi
- Open LabVIEW and select Tools » Microprocessor SDK » Target Editor.
- Select File » Open and browse to the LabVIEW\Targets\Keil\Embedded\RealView\AT91RM9200\ directory.
- Locate and open the TgtSupp.xml file.
- Right-click the AT91RM9200 icon and select Properties.
- Change the Path to Elemental I/O directory to eio\EIO and click OK.
- Select File » Save and exit the Target Editor.
- Exit LabVIEW completely and open LabVIEW again so that the targets are renumerated.

Digital Input/Output
The AT91RM9200 features our 32-bit PIO Controllers with Up to 122 Programmable I/O Lines, Input Change. Interrupt and Open-drain Capability on Each Line. The four controllers are named A, B, C and D and the schematics can be viewed in the appendix. This thesis will show how to program elemental I/O for one pin on one of these controllers, PIOA. There are no configurable properties for digital input/output; therefore, you need two Implementation VIs, one for input and one for output.

- Make a copy of the EIOImplementation – Read Scalar.vi to the LabVIEW\Targets\Keil\Embedded\RealView\AT91RM9200\EIO\GPIO directory and rename it AT91RM9200_Implementation_PIOA_wite.vi.
- Open AT91RM9200_Implementation_PIOA_wite.vi in LabVIEW. If some files can’t be found during the loading process first check if they are located in the \AT91Rm9200\EIO directory after that search in the LabVIEW directory after them.
- On the block diagram, delete all of the comments and the following controls and indicators:
  - Project attributes
  - Node attributes
  - IO Name (original)
  - IO Name (alias)
- On the front panel, right-click the Data indicator and replace it with a Boolean control. Do not delete the Data indicator and place a Boolean control, or the terminal placement will be incorrect.
- There should be two error terminals: a Boolean control and an I8 control on the block diagram.
• The actual code in the elemental I/O is different on different microcontrollers and can be solved in numerous ways, figure 10 shows one solution.
• Be sure to add the AT91RM9200.h and lib_AT91RM9200.h header files to the Inline C Node by right-clicking the Inline C Node border and selecting Add or Remove Header Files....
• Click Add File and browse to the /Keil/ARM/Inc/ATMEL/RM9200 directory and select the two header files. Click OK twice to return to the block diagram.
• The Implementation VI for digital output is now complete.

![Figure 10: VI describing the PIOA elemental output](image)

To make the VI for digital input copy the AT91RM9200_Implementation_PIOA_wite.vi and rename it to AT91RM9200_Implementation_PIOA_read.vi.

Open AT91RM9200_Implementation_PIOA_wite.vi in LV and do the changes shown in figure 11:
Figure 11: VI describing the PIOA elemental input

Note: Consult the hardware manual for your target to determine the register-level programming necessary to configure and access peripherals and the I/O hardware.

Creating Elemental I/O Devices, Classes, Pins and Resources

When all the Vis describing the Elemental I/O:s has been completed the next step is to implement them in LabVIEW using the Elemental I/O Device Wizard. This tutorial will give guidance of how to make the two

- Open LabVIEW and select Tools>>ARM module>>Elemental I/O Device Wizard.
- Enter AT91RM9200 in the I/O Device name and click next

- Next add the elemental I/O classes that you would like to use
- In this case only the Digital input/output will be used so that is what should be selected in the next step. But if the user tends to use more classes than this they should be selected as well.
• When adding I/O Resources make sure that you use proper Resource prefixes and consult the manual for the correct index numbers.

• After adding all the I/O Resources a list will show all the different I/O:s you want to use.

• In this particular case there will be no use for Pins so click next three times until the “Add Implementation Vis to Resources” window appears.
• Select PIA0 and click on Edit write methods.
• Add the AT91RM9200_Implementation_PIOA_Write.VI and click OK
• Select POA0 and click on Edit read methods.
• Add the AT91RM9200_Implementation_PIOA_Read.VI and click OK
• After all your implementation Vis have been assigned to the correct Elemental I/O click next
• Choose the correct target and output directory, C:\Program Files\National Instruments\LabVIEW 8.6\Targets\Keil\Embedded\RealView\AT91RM9200\eio\EIO
• Click Generate, and LabVIEW creates the files for implementing elemental I/O in the selected Output Directory

Using the elemental I/O nodes

To use the created elemental I/Os the user first needs to start a new project in LabVIEW. Then adding the correct ARM target by right clicking on Project and select NEW>>Target and Devices and choosing the created target.
Then users create elemental I/O items in the Project Explorer window by right-clicking a VI and selecting New»Elemental I/O from the shortcut menu. Users can then drag the elemental I/O item from the Project Explorer window to the block diagram.
Users configure an Elemental I/O Node by right-clicking the node on the block diagram and selecting Properties from the shortcut menu. Users select configuration parameters you implement in the elemental I/O implementation VIs.

### AT91RM9200 Schematics

**PAx**

<table>
<thead>
<tr>
<th>PIN</th>
<th>Signal Name</th>
<th>PIN</th>
<th>Signal Name</th>
<th>PIN</th>
<th>Signal Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>G13</td>
<td>PA0</td>
<td>B15</td>
<td>PA11</td>
<td>D12</td>
<td>PA22</td>
</tr>
<tr>
<td>G14</td>
<td>PA1</td>
<td>B16</td>
<td>PA12</td>
<td>A12</td>
<td>PA23</td>
</tr>
<tr>
<td>G15</td>
<td>PA2</td>
<td>A17</td>
<td>PA13</td>
<td>B12</td>
<td>PA24</td>
</tr>
<tr>
<td>G16</td>
<td>PA3</td>
<td>A15</td>
<td>PA14</td>
<td>E13</td>
<td>PA25</td>
</tr>
<tr>
<td>F13</td>
<td>PA4</td>
<td>B14</td>
<td>PA15</td>
<td>E12</td>
<td>PA26</td>
</tr>
<tr>
<td>F16</td>
<td>PA5</td>
<td>D14</td>
<td>PA16</td>
<td>G12</td>
<td>PA27</td>
</tr>
<tr>
<td>E15</td>
<td>PA6</td>
<td>B13</td>
<td>PA17</td>
<td>F12</td>
<td>PA28</td>
</tr>
<tr>
<td>B17</td>
<td>PA7</td>
<td>C13</td>
<td>PA18</td>
<td>E11</td>
<td>PA29</td>
</tr>
<tr>
<td>C16</td>
<td>PA8</td>
<td>A13</td>
<td>PA19</td>
<td>F11</td>
<td>PA30</td>
</tr>
<tr>
<td>E14</td>
<td>PA9</td>
<td>C12</td>
<td>PA20</td>
<td>A10</td>
<td>PA31</td>
</tr>
<tr>
<td>D15</td>
<td>PA10</td>
<td>D13</td>
<td>PA21</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PBx**

<table>
<thead>
<tr>
<th>PIN</th>
<th>Signal Name</th>
<th>PIN</th>
<th>Signal Name</th>
<th>PIN</th>
<th>Signal Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>B10</td>
<td>PB0</td>
<td>F9</td>
<td>PB11</td>
<td>C4</td>
<td>PB22</td>
</tr>
<tr>
<td>C5</td>
<td>PB1</td>
<td>C8</td>
<td>PB12</td>
<td>F3</td>
<td>PB23</td>
</tr>
<tr>
<td>C4</td>
<td>PB2</td>
<td>C7</td>
<td>PB13</td>
<td>F5</td>
<td>PB24</td>
</tr>
<tr>
<td>E10</td>
<td>PB3</td>
<td>D7</td>
<td>PB14</td>
<td>F4</td>
<td>PB25</td>
</tr>
<tr>
<td>A9</td>
<td>PB4</td>
<td>E7</td>
<td>PB15</td>
<td>G4</td>
<td>PB26</td>
</tr>
<tr>
<td>B9</td>
<td>PB5</td>
<td>F7</td>
<td>PB16</td>
<td>G6</td>
<td>PB27</td>
</tr>
<tr>
<td>C9</td>
<td>PB6</td>
<td>A4</td>
<td>PB17</td>
<td>J5</td>
<td>PB28</td>
</tr>
<tr>
<td>E9</td>
<td>PB7</td>
<td>B3</td>
<td>PB18</td>
<td>J6</td>
<td>PB29</td>
</tr>
<tr>
<td>D9</td>
<td>PB8</td>
<td>C5</td>
<td>PB19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A8</td>
<td>PB9</td>
<td>A3</td>
<td>PB20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B8</td>
<td>PB10</td>
<td>B4</td>
<td>PB21</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PCx**

<table>
<thead>
<tr>
<th>PIN</th>
<th>Signal Name</th>
<th>PIN</th>
<th>Signal Name</th>
<th>PIN</th>
<th>Signal Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>M15</td>
<td>PC0</td>
<td>N16</td>
<td>PC11</td>
<td>U17</td>
<td>PC22</td>
</tr>
<tr>
<td>L12</td>
<td>PC1</td>
<td>N17</td>
<td>PC12</td>
<td>T16</td>
<td>PC23</td>
</tr>
<tr>
<td>P14</td>
<td>PC2</td>
<td>M13</td>
<td>PC13</td>
<td>R15</td>
<td>PC24</td>
</tr>
<tr>
<td>L15</td>
<td>PC3</td>
<td>M17</td>
<td>PC14</td>
<td>T17</td>
<td>PC25</td>
</tr>
<tr>
<td>K14</td>
<td>PC4</td>
<td>L13</td>
<td>PC15</td>
<td>P14</td>
<td>PC26</td>
</tr>
<tr>
<td>K13</td>
<td>PC5</td>
<td>U14</td>
<td>PC16</td>
<td>P15</td>
<td>PC27</td>
</tr>
<tr>
<td>K15</td>
<td>PC6</td>
<td>R13</td>
<td>PC17</td>
<td>N13</td>
<td>PC28</td>
</tr>
<tr>
<td>T4</td>
<td>PC7</td>
<td>U15</td>
<td>PC18</td>
<td>R16</td>
<td>PC29</td>
</tr>
<tr>
<td>R5</td>
<td>PC8</td>
<td>T14</td>
<td>PC19</td>
<td>N15</td>
<td>PC30</td>
</tr>
<tr>
<td>P7</td>
<td>PC9</td>
<td>U16</td>
<td>PC20</td>
<td>N14</td>
<td>PC31</td>
</tr>
<tr>
<td>L12</td>
<td>PC10</td>
<td>T15</td>
<td>PC21</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PDx**

<table>
<thead>
<tr>
<th>PIN</th>
<th>Signal Name</th>
<th>PIN</th>
<th>Signal Name</th>
<th>PIN</th>
<th>Signal Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>E17</td>
<td>PD0</td>
<td>A5</td>
<td>PD11</td>
<td>H3</td>
<td>PD22</td>
</tr>
<tr>
<td>F15</td>
<td>PD1</td>
<td>B5</td>
<td>PD12</td>
<td>H1</td>
<td>PD23</td>
</tr>
<tr>
<td>F14</td>
<td>PD2</td>
<td>B2</td>
<td>PD13</td>
<td>J3</td>
<td>PD24</td>
</tr>
</tbody>
</table>
Code Optimization

Optimization techniques are important in embedded programming. There are two main reasons for optimizing embedded code. One is to optimize the code for speed. This is important when using high-frequency loop rates. Another reason to optimize code is for space utilization on the ARM microprocessor.

There are several optimization techniques to turn a non-functioning or poorly functioning application into a workable solution. The table below lists some of the main optimization techniques and how they affect embedded speed and size.

<table>
<thead>
<tr>
<th>Optimization Technique</th>
<th>Speed</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use small data types</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Avoid large Vis</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Use non-reentrant sub Vis</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Use reentrant sub Vis</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Use parallel operations</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Use single-cycle Timed Loops</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Use pipelining</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Use appropriate arbitration</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Various C-functions from LabVIEW vs C test

FFT:
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include "fftw3.h"
//size_t runs, size_t sz
int main(int argc, char* argv[])
{
    int status = 0;
    if(argc == 3)
    {
        clock_t start;
        double elapsed;
        size_t runs = (size_t)atoi(argv[1]), sz = (size_t)atoi(argv[2]);
        size_t run, i;
        fftw_complex *in;
            fftw_complex *out;
            fftw_plan plan_forward;
        //Create the input array.
            in = (fftw_complex*)fftw_malloc(sizeof(fftw_complex) * sz);
            for (i = 0; i < sz; ++i)
                {
                    in[i][0] = i;
                    in[i][1] = 0.00;
                }
        //Start timing
        start = clock();
        //Create the output array
            out = (fftw_complex*)fftw_malloc(sizeof(fftw_complex) * sz);
            plan_forward = fftw_plan_dft_1d(sz, in, out, FFTW_FORWARD, FFTW_ESTIMATE);
        //Perform FFTs
            for(i = 0; i < runs; ++i)
                fftw_execute(plan_forward);
        //Calculate elapsed time
        elapsed = clock() - start;
        printf("Elapsed Time: %.4fms\nTime Per Iteration: %.4fms", elapsed, elapsed / runs);
        //Clean up
        fftw_destroy_plan(plan_forward);
        fftw_free(in);
        fftw_free(out);
    }
    else //syntax error
    {
        status = -1;
        printf("Usage: FFT [Iterations] [Data Size]");
    }
    return status;
}
FILEWrite

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#pragma warning(disable : 4996)  //deprecation warning for fopen

//size_t runs, const char* fname
int main(int argc, char* argv[])
{
    int status = 0;

    if(argc == 3)
    {
        clock_t start;
        double elapsed;
        size_t i, runs = (size_t)atoi(argv[1]), size = 1000000;
        const char* fname = argv[2];
        char* data = (char*)malloc(size*sizeof(char));
        memset(data, 65, size);
        //open file
        FILE* outFile = fopen(fname, "wb");
        if(outFile)
        {
            //start timing
            start = clock();
            for(i = 0; i < runs && !status; ++i)
            {
                status = size != fwrite(data, sizeof(char), size, outFile);
            }
            //calculate elapsed time
            elapsed = clock() - start;
            printf("Elapsed Time: %.3lfms\nTime Per Iteration: %.3lfms", elapsed, elapsed / runs);
            //release file
            fclose(outFile);
        }
        //cleanup
        free(data);
    }
    else    //syntax error
    {
        status = -1;
        printf("Usage: FileWrite [Iterations] [Filename]\n");
    }
    return status;
}