J. Beningo, *Embedded Software Design*
https://doi.org/10.1007/978-1-4842-8279-3_11

11. Selecting Microcontrollers

Jacob Beningo[1]
(1) Linden, MI, USA

Selecting the right microcontroller for the job can be a stressful and disconcerting endeavor. Take a moment to perform a microcontroller search at Digi-Key, Mouser, or your favorite parts distributor. While I write this, Digi-Key has over 90,000 microcontrollers available in their catalog from more than 50 different manufacturers! I suspect that there are still far more than this; it's just that Digi-Key does not have them on their line card.

The sheer number of microcontrollers available is overwhelming. So how does one go about finding a microcontroller that fits their application? That, dear reader, is precisely what we are going to discuss in this chapter.

The Microcontroller Selection Process

When I first started to develop embedded software professionally in the early 2000s (yes, I was a maker in the mid-1990s before being a maker was a thing), selecting a microcontroller was simple; we let the hardware engineer do it! Microcontroller selection was left to the hardware engineer. At some point, in the design process, there would be an excellent unveiling meeting where the software engineers would learn about the processor they would be writing code for.

Software developers, at least in my experience, had little to no input into the microcontroller selection process. If we were lucky, the hardware designers would ask for feedback on whether the part was okay for software. In many cases, the unveiling of the hardware wasn't just schematics or a block diagram; it was the meeting where prototype hardware was

provided to the developers. The modern selection process is quite different and emphasizes the software ecosystem (at least it should!). Failure to consider the software needs often results in a poor solution with a high chance of never making it to market.

The modern microcontroller selection process contains seven significant steps:

1)Create a block diagram of the hardware
2)Identify all the system data assets
3)Perform a threat model and security analysis (TMSA)
4)Review the software model and architecture
5)Research microcontroller and software ecosystems
6)Evaluate development boards
7)Make the final microcontroller selection

These steps may seem obvious or entirely foreign, depending on how experienced you are in selecting processors. Let's examine each of these steps in more detail so that you can choose the right microcontroller on your next project. On the other hand, you may find that some of these steps are pretty familiar and correspond to the design philosophies we discussed in Chapter 1.

Step #1 – Create a Hardware Block Diagram

The first step toward selecting a microcontroller, and probably for any step in developing an embedded system, is to create a hardware block diagram. The block diagram helps constrain the design and tells the engineer what they are dealing with. A software architecture can't be developed without one, and the microcontroller can't be selected without one either!

A block diagram serves several purposes in microcontroller selection. First, it identifies the external components connected to the microcontroller. These components could be anything from simple GPIO devices like push buttons or relays to more complex analog and digital sensors. Second, it identifies the peripherals needed to communicate with those external components. These could be peripherals like I2C, SPI, USART,

USB, etc. Finally, the block diagram provides a minimum pin count that is required. This pin count can be used in online part searches to filter for parts that don't provide enough I/O.

> **Best Practice**Don't forget to include a few spare I/O for future hardware expansion and for debugging such as for toggling for timing measurements.

An early analysis of the hardware diagram can help to identify potential issues and shortcomings in the system. The analysis can also help to determine if the design is too complex or if there could be other business-related issues that could prevent the product from being completed successfully. An example block diagram for a smart thermostat can be seen in Figure 11-1. Notice that the block diagram segments the different external components based on the peripheral that will be used to interact with it. I will often create diagrams like this and add the pin count required for each to understand the I/O requirements. For example, Figure 11-1 currently has identified at least 23 I/O pins needed to connect all the external devices.

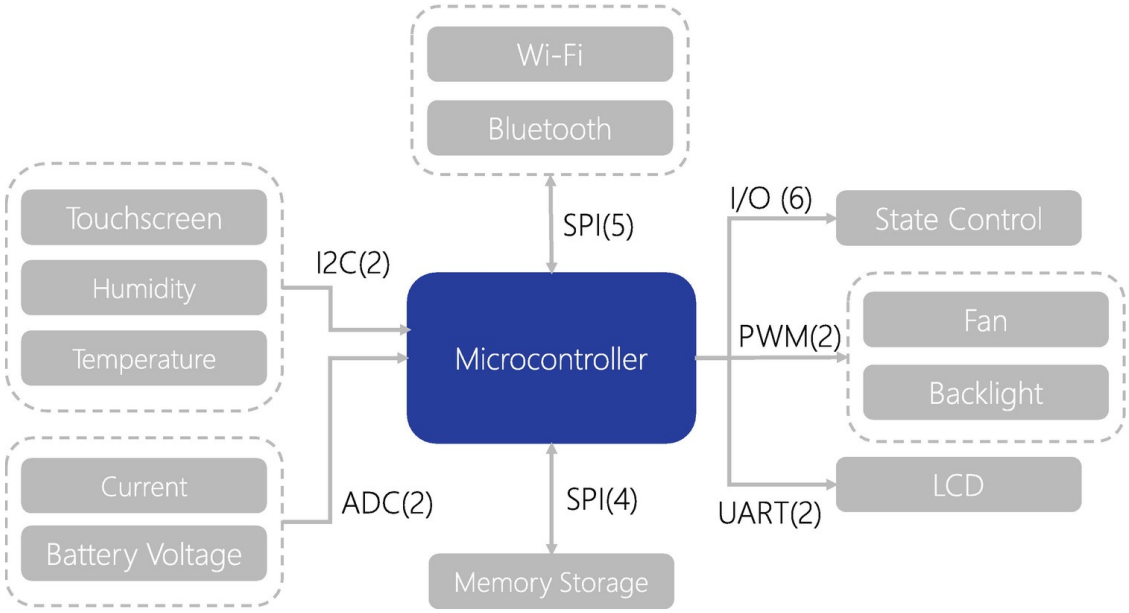

Figure 11-1 An example block diagram identifies significant system components and bus interfaces used to identify microcontroller features and needs

Peripherals connected to external components aren't the only peripherals to consider when creating the block diagram. There are internal peripherals that can be just as important. For example, I've often found it helpful to develop a microcontroller-centric block diagram like that shown in Figure 11-2. This diagram identifies how many pins each peripheral uses in addition to internal peripheral needs. Internally, a microcontroller may need features such as

- Direct memory access (DMA)
- Cryptographic accelerators
- Watchdog timers
- Arm TrustZone
- Memory protection unit (MPU)
- Enhanced debug capabilities like the ETM

It's helpful to place these onto a block diagram to ensure that they are not overlooked when it is time to research and select a microcontroller. Otherwise, there may be too much focus on the external peripherals, and essential design features may be overlooked.

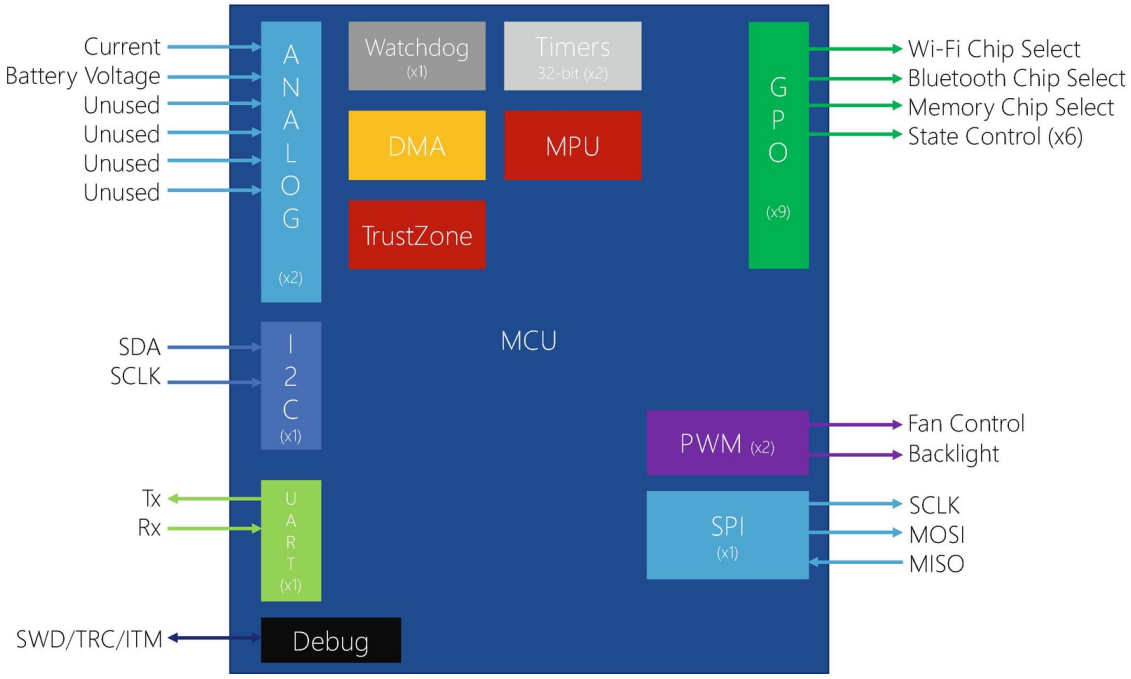

Figure 11-2 The peripherals and pins needed for an application can be broken down into a simple block diagram that shows the microcontroller requirements in a single, easy-to-digest image

Once we understand the external and internal peripheral sets, we have identified important constraints on the microcontroller selection process. We know what hardware features our microcontroller needs, but we still don't know how much processing power the microcontroller needs. To understand the processing needs, we need to identify our data assets.

Step #2 – Identify All the System Data Assets

Well, here we are again! We once again need to identify the data assets in our system! I hope this theme is helping you to understand just how im-

portant it is in embedded software and system design to focus on the data. As you should recall from Chapter 1, data dictates design! Selecting a microcontroller is part of the design, even if it is the hardware component of the design.

We want to start to identify the data assets at this point in the microcontroller selection process because the data will help a developer properly scope and size their software. The software required to achieve the desired system behavior will dictate whether a team can use a resource-constrained Arm Cortex-M0+, Cortex-M23, or something with more performance like an Arm Cortex-M7, Cortex-M33, or Cortex-M55 is required. Identifying the data assets used in the system can then be fed into a threat model security analysis (TMSA) which we perform in Step #3.

Identifying data assets can be a straightforward process. I often start by just creating a list of data. Then, I look at the system's inputs and outputs and make a bullet list in a word document. Once I have decomposed the system into the data assets, I create a simple table that lists the data and then try to fill in some operational parameters. Parameters that I often list include

- The type of data asset
- Data size
- Sample rate
- Processing requirements

At this stage, it's important to note that there is more to data assets than simply the operational inputs and outputs. The firmware itself is a data asset! Any certificates, public/private keys, and so forth are all data assets! You want to make sure that you identify each data asset. We often use terminology such as data in-flight (or in-transit) to indicate data that moves into, out of, and within the system. Data at rest is data that doesn't move but can still be more vulnerable to someone interested in attacking the system.

Table 11-1 shows an abbreviated example of what a data asset table might look like. The table lists the data assets that I've identified for the smart thermostat, the type of data it is, and several other parameters that would be pretty useful in scoping the system. Note that this is not an activity in figuring out software stacks

that will be used, digital filtering that would be applied, and so forth. It's just to identify the data assets and get a feel for how much processing power and software may be required.

Table 11-1 This is a sample data asset table. It has been abbreviated to minimize the space that it takes up. However, it provides a general overview of what a developer would want to list to identify data assets in their system

Data Asset	Asset Type	Data Size	Sample Rate	Processing
Analog	Sensor	32 bytes	1 kHz	Digital notch filter
Digital	Sensor	128 bytes	1 kHz	Running average – 5 sample
Firmware	IP	256 Kbyte	–	See design documentation
Keys	Keys	128 bytes	–	Secure storage
Device ID	Data	128 bits	–	Secure storage

Step #3 – Perform a TMSA

Once the system data assets have been listed, it's time for the team to perform a threat model and security analysis (TMSA). The idea behind a TMSA is that it allows a team to evaluate the security threats that their system may face. A key to performing that analysis is to first list the system data assets, which was exactly what we did in Step #2. Next, we perform a TMSA, and do so early in the design cycle, because it provides us with the security objectives we need to design our software and select a microcontroller.

It's essential to recognize that security cannot be added to a system at the end of the development cycle. Like quality, it must be built into the system

from the beginning. The TMSA will provide the security requirements for the system. If security is not needed, then the TMSA will be very fast indeed! Honestly, though, every system has some level of security. At a minimum, most businesses don't want a competitor to be able to pull their firmware from the device so they can reverse-engineer it. They often need secure firmware updates so a system can't have other firmware placed on it. When someone tells me that they don't need security, it just indicates that they don't know what they need and that I'll have to work extra hard to educate them.

The details about designing secure firmware and performing a TMSA can be found in Chapter 3. I won't rehash the details here, but if you are skipping around the book and want to understand the details, it may be worthwhile to pause and go back and read that chapter. Let's discuss the fourth step in the microcontroller selection process for the rest of us.

Step #4 – Review the Software Model and Architecture

From the very beginning, software designers are hard at work figuring out what the software architecture looks like and how the software will behave. It's critical that the software be designed before the final microcontroller is selected! Without the software architecture, the chosen processor could be far too powerful for the application at hand, or, worse, it's possible that it won't have enough processing power for the application!

Now, you don't necessarily want to perfectly pair your microcontroller with the software you need to run. Instead, you generally want to have a little more processing power, memory, and capability than you need. The reason for this is that products often scale and grow over time. So if you pick the perfectly sized processor at launch and then try to add new features over the next several years, you may end up with a slow system or something that causes customers to complain about.

However, we need to figure out the right processing power, memory, and capabilities based on the software we need to run. We shouldn't just pick a processor because we like it or think it should work. Instead, we need to evaluate our software model and architecture and select the right proces-

sor based on the software needs. Yes, software dictates what hardware we need, NOT the other way around. In the “old days,” that’s exactly how things used to be done.

Typically, at this stage, I identify the software stacks and components that I need to meet the system's objectives. For example, if I know I have an Ethernet stack or USB, I want to list that out. That may require a specialized part, or it may lead me to decide I want to use multiple processors or even a multicore processor. The goal is to get a feel for what is needed to run the various software stacks and the stacks and middleware that will be required. Once we understand that, we can start researching different microcontrollers and the software ecosystems surrounding them!

Step #5 – Research Microcontroller Ecosystems

I feel that over the past several years, the differentiation between different microcontrollers hasn’t come from the physical hardware but the software ecosystems surrounding them. The hardware is standard and agnostic. Most vendors have moved to an Arm Cortex-M core, although some proprietary cores still exist, and RISC-V has been slowly gaining traction in some circles. The differences in hardware between vendors, though, are not terribly dramatic. Even though 32-bit microcontrollers have been coming to dominate the industry, sometimes the best match will be a little 8-bit part like a Microchip PIC processor. In our analysis, we shouldn’t immediately rule these out.

Most GPIO peripherals have the same or very similar features. USARTs, SPI, PWM, and many other peripherals also provide similar capabilities. The registers to configure them are often different, but it doesn’t matter whether I go with microcontroller Vendor A or Vendor B. In fact, the hardware capabilities are so standard that industry-wide hardware abstraction layers (HALs) have popped up to ease migration between vendors and software portability. (Note that I am exaggerating a bit; there are plenty of parts that are designed for specialized applications, but there are a lot of general-purpose parts out there too.)

The fundamental difference between microcontrollers is in the ecosystem

that the vendor provides! In a world where we are constantly working with short delivery times, limited budgets, and so forth, the microcontroller's ecosystem can be the difference between whether we deliver on time with the expected quality or are over budget over time shipping junk. The ecosystem is the differentiator and probably should be one of the most significant factors we use as developers to select a microcontroller.

The ecosystem encompasses everything that supports the microcontroller, including

- Driver and configuration tools
- Real-time operating systems (RTOS)
- Middleware stacks
- Modeling and code generation tools
- Compilers and code optimizers
- Programming, trace, and debug tools
- The developer community and forums that support developers

The ecosystem can even include application examples and anything that makes the developers' job more manageable.

I once had a customer who came to me with his hardware already selected and developed. He just wanted someone to write the software. Upon examination of the hardware, I discovered the customer had chosen a microcontroller that was only available with a six-month lead time, was not supported by significant tool manufacturers, and had no example code or supporting forums because no one was using the part. Furthermore, they had selected their microcontroller poorly! As you can imagine, everything in that project was an uphill battle, and there was no one to turn to when things didn't work as expected.

Don't put yourself in that situation! Instead, select widely adopted, supported, available microcontrollers that have a rich ecosystem.

Step #6 – Evaluate Development Boards

One of my favorite parts of the microcontroller selection process is purchasing development boards for evaluation. At this point in the project, we've usually gone through the due diligence of the design and are ready to start working on the low-level software. In some cases, the high-level business logic of the application has already been developed, and we are just missing the embedded processor. In other cases, we are evaluating and starting with the low-level firmware to prove any high-risk areas and help the hardware team develop their piece in parallel.

Evaluating development boards can also fill in design gaps. For example, it's good to put blocks in a design for MQTT, Wi-Fi, and cloud connectivity modules, but no team will write these themselves. Instead, the selected ecosystem and vendor will dictate these design blocks. In addition, running example software and software stacks on the development board can help pin down what modules, tasks, and timing are required. These elements, once known, can require adjustments to the design to ensure the system works as expected.

When evaluating development boards, there are generally a few tips I like to follow. First, I recognize that the development board microcontroller is likely far more powerful than the microcontroller I will select. An evaluation board is usually the fastest and best processor in that chip family. The processor will have the most RAM, ROM, and so forth. The quicker I can build a rapid prototype of my application on the evaluation board, the better idea I can get about which part in the microcontroller family will best fit my application.

Chances are that nearly an unlimited number of vendors could supply a microcontroller for the desired application. So the first thing I do is pick two potential vendors I already have experience with. I'll then look for a third that I don't have experience with, but that looks like they have an ecosystem that fits our needs. I'll then look through their offerings to identify several microcontrollers that could work and build up a list like Table 11-2 that lists out the development boards that are available for evaluation.

Table 11-2 This example table lists development boards that might be evaluated for an IoT application. Again, this is truncated for space but would include additional details like the MCU part #, price, lead times, and checkboxes for whether the board met specific requirements

Vendor	Development Board	Clock Speed (MHz)	RAM (KB)/ROM (KB)	Features
STM	L4S51-IOT01A	120	640/2048	Wi-Fi, sensors, etc.
STM	IDW01M1	–	64/512	Wi-Fi, sensors, etc.
Microchip	SAM-IoT	20	6/48	Wi-Fi, security, etc.

An important point to note is that I don't look at the price of the evaluation boards. They will range from $20 to $600. I decide which one has everything I need to make my evaluation. Suppose that is the $20 board, awesome. If it's the $600 board, that's perfectly fine. I don't let the price get in the way. Sure, if I can save $500, that's great, but I'd rather spend the $500 if it will shave days off my evaluation or, more importantly, help me to select the correct part. Choosing the wrong part will cost more than $500, so I'd instead do it right. (Too many developers get hung up on the price for boards, tools, training, etc. It may seem like a lot to the developer, but these costs don't even make the decimal place on the balance sheet to the business. So don't cut corners to save a buck in the short term and cost the business tens of thousands later.)

Once the selected development boards arrive, spend a few days with each to get the main components and examples up and running. Evaluate how well the base code and configurations fit your application. Use trace tools to evaluate the architectural components that are already in place. Explore how the existing code could impact your architecture. Take some time to run out to the forums and ask questions and see how quickly you can get a response. Get the contact information for the local Field Application Engineer (FAE) and introduce yourself and ask questions again. Again, check and see how quickly they respond. (Don't base too

much of your decision on response time, though. If you are only building 1000 units, the forum will be where you get most of your support. The FAE will do their best, but they often are pulled in many directions and must respond to the customers by building millions or hundreds of thousands of units first.)

Step #7 – Make the Final MCU Selection

The last step is to make the final selection on the microcontroller, which one seems to fit best for the application. In most cases, this is not a life-or-death decision. Many systems can be built successfully no matter which microcontroller is selected. What is really on the line is whether the microcontroller and ecosystem will match the product and business needs. Unlike at school or college, there isn't necessarily a right or wrong answer. It's finding the best fit and then going for it!

Our discussion on selecting a microcontroller has been high, with some tips and tricks scattered throughout. You might wonder how you can be certain that the selected part will be a good fit for your product and the company you work for. If you are working on a team, you'll discover that every engineer will have their own opinion on which part should be used. In many cases, personal bias can play a massive role in the microcontroller selection process. Let's now examine a process I use myself and my clients to help remove those biases and select the right microcontroller for the application.

The MCU Selection KT Matrix

Whether or not we want to admit it, our personal bias can play a significant role in our decision-making processes. For example, if I have used STM32 microcontrollers in the past with ThreadX successfully, I'm going to be more biased toward using them. If I am an open source junky, I may push the team to avoid using a commercial RTOS like uC OS II/III or VxWorks, even if they are the right solution for the project. When selecting a microcontroller or making any major team decision, we need to try to remove personal bias from the decision. Now don't get me wrong,

some personal bias can be good. If I'm familiar with the STM32 parts, that can be an asset over using a part I am completely unfamiliar with. However, we do want to remove unmanaged bias that doesn't benefit the microcontroller selection process. This is where a KT Matrix can make a big difference.

A KT Matrix, more formally known as the Kepner Tregoe matrix, is a step-by-step approach for systematically solving problems, making decisions, and analyzing potential risks.[1] KT Matrixes limit conscious and unconscious biases that steer a decision away from its primary objectives. The KT Matrix has been used to make organizational decisions for decades, but while it is a standard tool among project managers, embedded engineers rarely have been exposed to it. We're going to see precisely how this tool can be used to help us select the right microcontroller for our application.

Identifying Decision Categories and Criterions

The idea behind a KT Matrix is that we objectively identify the decision we want to make. Therefore, we start by defining the problem. For example, we want to decide which microcontroller seems to be the best fit for our application. To do this, we need to identify the criteria used to make the decision and determine the weights each criterion has on our decision-making process. For example, I may decide to evaluate several categories in my decision-making, such as

- Hardware
- Features
- Cost
- Ecosystem
- Middleware
- Vendor
- Security
- Experience

These are the high-level categories that I want to evaluate. Each category will have several criteria associated with it that help to narrow down the decision. For example, the middleware category may include

- RTOS
- File system
- TCP/IP networking stacks
- USB library
- Memory management

When I identify each criterion, I can apply weight to how important it is to my decision. I will often use a scale of one to five, where one is not that important to the decision and five is very important. I'm sure you can see that the weight for each criterion can easily vary based on opinion. We all know that from a business perspective, management will push the cost to be a rating of four or five. To an engineer, though, the price may be a two. The KT Matrix is powerful in that we can get the stakeholders together to decide on which categories and criteria will be evaluated and how much weight each should have!

Building the KT Matrix

Once the categories and criteria have been decided, a KT Matrix can be developed using a simple spreadsheet tool like Excel or Google Sheets. When building the KT Matrix, there are several columns that we want to make sure the spreadsheet has, such as

- Criterion
- Criterion weight (1–5)
- Microcontroller option (at least 3)

Each microcontroller column is also broken up to contain a column for each person who will help decide whether the microcontroller fits the application or not. Each person rates on a scale from one to five how well the criterion meets the product's needs. These are then all weighted and summed together for each team member. Figure 11-3 shows an abbreviated example of how the microcontroller KT Matrix looks using just a few categories.

			Microcontroller #1						Microcontroller #2					
	Criteria	Weight	Rating 1	Rating 2	Rating 3	Rating 4	Rating 5	Weighted Rating Total	Rating 1	Rating 2	Rating 3	Rating 4	Rating 5	Weighted Rating Total
Hardware	32-bit Architecture	4						0						0
	Processor speed	4						0						0
	Instruction set	5						0						0
	Minimial interrupt latency	5						0						0
	Loweset energy consumption	5						0						0
	Part Availability	5						0						0
	Memroy footprint / speed	4						0						0
Features	Best Real-time trace capabilities (ITM, ETM)	3						0						0
	Memory protection unit (MPU)	4						0						0
	FPU	4						0						0
	Driver and middleware configuration tools	5						0						0
	Safety certifications	5						0						0
	Hardware accelerated cryptography	5						0						0
	Multicore	3						0						0
	RTOS Support	3						0						0
	Wireless Connectivity	4						0						0
Cost	Lowest upfront licensing costs	5						0						0
	Lowest royalty cost per unit	3						0						0
	Smallest tool investment	4						0						0
	Lowest training investment	5						0						0
	Lowest cost of middleware (price and integration effort vs quality)	5						0						0
	Least open source (minimize new IP release)	3						0						0

Figure 11-3 An example microcontroller selection KT Matrix shows how a team can evaluate how well a microcontroller fits their application unbiasedly

A more detailed example, and one that is more readable, can be downloaded and modified by the reader from beningo.com using the link: `https://bit.ly/3cL1P2X`. Now that the matrix is in place, let's discuss how we perform our analysis.

Choosing the Microcontroller

What I love most about the KT Matrix is that the decision-making process is reduced to nothing more than a numerical value! At the end of the day, the microcontroller that has the highest numerical value is the microcontroller that best fits the application for the decision makers! (After all, the results would differ for a different team.) Getting the numerical value is easy.

First, each team member reviews the KT Matrix and the microcontrollers. They go through the criterion and rate on a scale from one to five how well that criterion meets the team's needs. Next, once each team member has put in their ranking, each criterion is weighted and summed for each microcontroller. Remember, we applied weight to each criterion based on how important it was to our decision. For example, if cost is weighted as four, we sum each decision maker's cost rating and multiply by four. We do this for each criterion. Once all the criteria have been summed and weighted, we sum up each microcontroller's weight column. Finally, the microcontroller with the largest weighted sum is our choice! An example can be seen in Figure 11-4.

			Microcontroller #1						Microcontroller #2					
	Criteria	Weight	Rating 1	Rating 2	Rating 3	Rating 4	Rating 5	Weighted Rating Total	Rating 1	Rating 2	Rating 3	Rating 4	Rating 5	Weighted Rating Total
Hardware	32-bit Architecture	4	3	3	3	3	3	60	2	2	2	2	2	40
	Processor speed	4	2	2	2	2	2	40	1	1	1	1	1	20
	Instruction set	5	2	1	1	1	2	35	1	2	2	2	1	40
	Minimial interrupt latency	5	1	2	2	1	1	35	3	1	1	3	2	50
	Loweset energy consumption	5	1	1	1	1	1	25	2	2	2	2	2	50
	Part Availability	5	1	2	1	1	1	30	2	3	3	3	3	70
	Memroy footprint / speed	4	3	3	3	3	3	60	2	2	2	2	2	40
Middleware	File system best meets system requirements	4	2	1	2	2	1	32	3	2	3	3	1	48
	TCP/IP stack best meets system requirements	4	2	1	2	2	1	32	3	2	3	3	1	48
	USB stack best meets system requirements	4	2	1	2	2	1	32	3	2	3	3	1	48
	Graphics stack best meets system requirements	4	2	1	2	2	1	32	3	2	3	3	1	48
	Middleware requires minimal integration effort	4	2	1	2	2	1	32	3	2	3	3	1	48
	Additional 3rd party tools integrated seamlessly	3	1	2	1	2	1	21	2	3	2	3	2	36
Engineer	Maxmize professional growth potential	2	2	2	1	3	1	18	1	1	3	2	3	20
	Least amount of stress to implement	2	2	3	1	1	3	20	1	2	3	3	2	22
	Most fun / interesting	1	2	3	3	1	2	11	3	1	1	2	3	10
	Minimized labor intensity	3	1	2	3	1	3	30	2	3	1	2	1	27
	Least deadline constrained to get up to speed	2	2	1	2	1	3	18	3	2	3	2	1	22
	Most internal resources available	3	1	2	3	3	3	36	2	3	1	1	1	24
Security	Security Certified RTOS	5	2	2	1	3	1	45	3	3	2	1	2	55
	Supports Arm TrustZone	4	1	1	2	1	1	24	2	2	3	2	2	44
	Supports TF-M	5	1	1	1	2	2	35	2	2	2	3	3	60
	Secure OTA / Bootloader support	3	2	2	1	2	2	27	1	1	2	3	3	30
	Total	198	98	94	101	101	95	1852	104	113	109	116	102	2059
			Microcontroller #1						Microcontroller #2					

Figure 11-4 An example microcontroller selection KT Matrix that the team members have evaluated. In this example, we can see that microcontroller #2 best fits the application and team needs

Looking closely at Figure 11-4, the reader can see that the weighted sum for microcontroller #1 is 1852, while the weighted sum for microcontroller #2 is 2059. In this example, microcontroller #2 has a higher numeric value and is the better fit for our application and team.

The KT Matrix is an excellent tool because it lets us make a decision objectively! The selection process is not determined by a single engineer with the strongest personality or management pushing down on costs. At the end of the day, selecting the right microcontroller for the job can make or break budgets, time to market, and the entire business!

Overlooked Best Practices

There are several best practices associated with microcontroller selection that teams can very easily overlook. Failure to adhere to these best practices won't end the world, but it can make things more difficult. If the project is more complicated, deadlines will likely be missed, budgets will be consumed, and other nasty things may happen.

The first overlooked best practice is to size the microcontroller memory, RAM, and ROM so that there are more than the current application needs. Sizing the memory early can be really tough. Designers can look for microcontroller families that are pin compatible that allow different size

memories to be selected. Final microcontroller selection can then be pushed off toward the end of development when the true memory needs are known.

The reason for oversizing the memory is to leave room for future features. I can't remember a single product I've worked on, consulted on, or advised on that did not add features to the product after the initial launch. Customers always ask for more, and businesses always want to keep adding value to their offerings. During microcontroller selection, developers need to anticipate this scope creep!

The next overlooked best practice is to select a microcontroller that has extra pins that are unused. Products evolve, and having to choose a new microcontroller, rewrite firmware, and update a PCB can be expensive and time-consuming! Instead, select a microcontroller that leaves room to add new hardware features in the future. For example, these pins can be used to add new sensors, or they could also be used during development to debug the application code.

Another overlooked best practice is to select a microcontroller that either has a built-in bootloader or that is sized for you to add a bootloader. Bootloaders are fantastic for providing in-field update capabilities. But, again, so many products evolve after launch that it's essential to provide a mechanism to update the firmware. I can't tell you how many times I have had a client tell me that they don't need a bootloader, only to call me within a month after launch telling me they need help making one (despite my advice to have one from the start!).

Using a bootloader can also add constraints to the design. It's hard to design a robust bootloader, especially a secure one. In many cases, to do it right, you need to trade off risk by having twice the memory so that you can keep a backup copy. The price of the microcontroller can be much larger due to these memory needs. Bootloaders aren't trivial and should be looked at early in the development cycle as well.

The last best practice for selecting a microcontroller isn't about choosing a microcontroller! But it can save some headaches for developers needing

to upgrade their microcontroller. When laying out a PCB, leave extra space around the microcontroller. Then, if the microcontroller ever goes end of life, gets a long lead time, or needs to be upgraded, a developer may be able to save time and budget on the upgrade if the entire PCB doesn't need to be redesigned. This trick saved me several times, especially during the COVID era, when microcontrollers became very hard to acquire in the supply chain.

Final Thoughts

Selecting the right microcontroller can be a tough job! There are so many choices and considerations that the selection process can be dizzying. In this chapter, we've seen that we can follow a simple process to identify and evaluate the microcontrollers that may best fit our applications. In addition, designers can't forget that the main purpose of selecting a microcontroller is to reduce the risk of failure. Businesses at the end of the day are looking to go to market and make a profit! Being too stingy or trying to overoptimize and use a small part can be a disaster.

Choose carefully. Consider the options. Evaluate future needs and availability and then find the microcontroller that is right for your application.

Action Items
To put this chapter's concepts into action, here are a few activities the reader can perform to start applying microcontroller selection best practices to their application(s):

- Work through the seven steps to select a microcontroller for your design or a fictitious design.
  - How is this process different from the one you used in the past?
  - Does/did it help you select a microcontroller that was a closer fit for your application?
- List the top three challenges that you encounter when you select a microcontroller. Then, what can you do going forward to alleviate or minimize these challenges?
- Download the RTOS Selection KT Matrix and examine how the KT Matrix works. (Hint: You don't need to use all the criteria I have put

there! You can remove them, add more, or do whatever you need to select your microcontroller!)
- In the RTOS Selection KT Matrix, explore the results tab. You'll notice that I break out how much weight each category has on the decision-making process. Why do you think this is important to do?

These are just a few ideas to go a little bit further. Carve out time in your schedule each week to apply these action items. Even small adjustments over a year can result in dramatic changes!

Footnotes

1 www.valuebasedmanagement.net/methods_kepner-tregoe_matrix.html