

EMBEDDED DESIGN FOR IOT WITH RENESAS SYNERGY™

COURSE BOOK – Prof. Dr. rer. nat. Felix Hüning



RENESAS synergy™
Accelerate. Innovate. Differentiate.

2017.11

BIG IDEAS
FOR EVERY SPACE

CONTENTS

FOREWORD	04
PREFACE	05
1 INTERNET OF THINGS & INDUSTRY 4.0	06
1.1 Embedded Systems	07
1.2 Sensors & Actuators	09
1.3 Central processing unit & software	10
2 MICROCONTROLLER	11
2.1 Peripheral modules	12
2.2 ARM architecture	15
2.3 Renesas Synergy™ MCU family	16
S1 series	17
S3 series	18
S5 series	18
S7 series™	19
S7G2 MCU	20
3 STARTER KITS	26
3.1 S7G2 Starter Kit	27
4 ISDE	29
4.1 Renesas e²studio	30
5 BOARD SUPPORT PACKAGE	35
5.1 SSP BSP	35
6 HARDWARE ABSTRACTION LAYER	40
6.1 SSP HAL	43
7 RTOS	47
7.1 SSP RTOS	54
8 APPLICATION FRAMEWORK & FUNCTIONAL LIBRARIES	58
8.1 SSP application framework and functional libraries	60
9 MIDDLEWARE	63
9.1 SSP middlewares	63
10 CONNECTIVITY	68
10.1 SSP connectivity	71
10.1.1 I²C	72
10.1.2 Ethernet	73
10.1.3 USB	76
11 RENESAS SYNERGY PLATFORM	78
12 CONNECTIVITY	83
12.1 Internet of Things & Industry 4.0	85
12.2 Microcontroller	86
12.3 Starter Kits	87
12.4 ISDE	87
12.5 Board Support Package	90
12.6 Hardware Abstraction Layer & RTOS	96
12.7 Framework & Functional Libraries	110
12.8 Connectivity	113
12.9 GUI	122
12.10 The Message Framework	136
REFERENCES	141
INDEX	142

FOREWORD

There is an exciting future ahead of us. Cars will drive autonomously and have sensory organs to detect the world around them. There will be farming on skyscrapers and buildings will self-generate energy. Robots will help in households or as medical assistants in remote diagnosis and surgeries. They will be even able to detect emotions, paint artworks or compose music.

The Internet of Things (IoT) will enable networks of connected devices, vehicles, buildings, and other embedded systems to collect and exchange data.

It will be you – the young, next generation of engineers - who will drive progress to a safer, healthier and greener world by developing creative solutions and revolutionary innovations that are far beyond our imagination today.

What do you need to realize your visions?

- Passion for the engineering sciences
- First class education
- State of the art technology systems – and practical experience how to use them
- Time for imagination and creativity
- Sense of responsibility for society, capability to handle human safety and data security

The Renesas Synergy™ Platform is an integrated system of microcontrollers and professional-grade software that has changed the way embedded systems are developed. It enables engineers to immediately start developing directly at the Application Programming Interface (API) level. This saves a lot of time for you - free time to get inspired and realize innovations!

This Synergy course book written by Prof. Felix Huening from the University of Applied Sciences in Aachen will show you step-by-step how to start developing code for your end application. At the same time you will learn about all the hardware and software components of embedded systems, and the benefits in ready to use solutions like a Board Support Package and Hardware Abstraction Layer.

Prof. Felix Huening is an ambassador between research, education and industry. He will pick you up from your individual knowledge level and guide you “step by step to high tech”. After completing the Synergy lab exercises you will be able to program your own application. Practising is best way to mastery!

On behalf of Renesas I would like to thank Felix for his outstanding commitment and contribution to the Renesas University Program. The close co-operation with him has resulted in many impressive students projects. My personal favourite was the Renesas RL78-based “outer space simulator” programed by a female student and delivered to us in a Zalando shoe box. Not only for this reason, Renesas appreciates diversity and we aim to help fascinate students of all genders in the Engineering sciences.

And now hands on to make the world a better place ...

Andrea Nuyken
Renesas Electronics Europe

PREFACE

Embedded systems are getting more and more important, not only for the high profile IoT and Industry 4.0 applications, but for all kinds of other application areas as well. You can find embedded systems in nearly every area, from simple white goods to much more complex systems such as those found in production plants and control systems. Hence the complexity of the applications increases in general and so does the complexity of the underlying hardware, like the microcontrollers. From an application point of view, it is highly desirable to focus on the application, not on the hardware. Of course, the hardware is essential and has to work in the required way, but in the end the main point is to develop an application or system – in the best case with a dedicated unique selling point! The question in the end is how to handle and combine these different complexities and how to find a smart way to make the use and programming of embedded systems in smart and complex applications, as simple and straightforward as possible.

Having this challenge for the development of embedded applications in mind it was a great opportunity for me when Renesas asked me to create a lecture for their new Renesas Synergy™ Platform. This platform allows users to start their development at a higher level than they could in the past. It converts the complexity of the hardware and the low level software into a high level approach to programming the corresponding MCUs. This higher level of abstraction makes the development of embedded applications much smarter, faster, simpler and more reliable. On the other hand, the complexity of the hardware and low level software is not hidden like a black box, but the user has full visibility of all drivers and hardware components, just in case he or she wants to work close to the hardware. The target of the Synergy Platform is to make the engineer work at a very high level of abstraction with a focus on the application. However, this approach can be difficult for students with little knowledge in embedded systems, application development and microcontroller systems. And it is difficult to get the complete picture just from a top view.

Therefore the aim of this coursebook is to support university lectures and labs, with the introduction of the concepts, components and tools in an embedded system development, from microcontroller basics to a high level of abstraction using the Renesas Synergy Platform as an example. As it enables high level development while maintaining full visibility of the complete system, it is an excellent example for educational purposes in the field of modern embedded system development. It offers the chance to cover the complete flow: start at the bottom, the microcontroller, experience the need for higher level components and modules, and finally understand the whole solution package provided by the Renesas Synergy Platform. The target group for this coursebook is end of bachelor or master level students of electrical engineering or similar. Prerequisites are a basic knowledge of microcontroller and embedded programming.

The organization of the book is as follows: each chapter starts with a general introduction and presentation of the corresponding basic topics. Afterwards the components of the Renesas Synergy Platform are introduced as an example. The book starts with a short introduction to the Internet of Things (IoT) and Industry 4.0 applications. The second and third chapter cover the hardware basics, microcontrollers as basic hardware components for embedded systems and Starter Kits. The development environment ISDE is introduced in chapter 4, followed by the low level drivers of the Synergy Software Package, the BSP and the HAL in chapters 5 and 6 respectively. As real-time behaviour is getting more and more important for embedded systems, chapter 7 deals with real-time operating systems. High level software components like frameworks and middleware are introduced in chapters 8 to 10. Finally, chapter 11 closes the theory part of the book, summarizing the features of the Renesas Synergy Platform.

Besides all the basics and theory, the practical work is the key to providing a full understanding. The lab described in chapter 12 uses one of the Synergy Starter Kits to get you started – right from the beginning! The structure of the lab is similar to the theoretical part of the coursebook. So just practice the corresponding part after each chapter – and finally get a smart home application running at the end!

This book would not have been possible without the support by many people. First of all, thanks to Patrick Zgoda for the development of the lab part, and many ideas and discussions about the Synergy Platform from a user's point of view. Giancarlo Parodi and Karol Saja reviewed the material and provided great feedback that improved the quality of the book significantly – thanks a lot! Also a great thanks to Steve Norman for proof reading and Andy Harding and the whole Synergy team for giving me the chance to work on this project – it really was a pleasurable and an interesting time! Klaus Rueschhoff and his Marcom Team converted the more or less unformatted text into the nice book you are reading now. And last but not least to Andrea Nuyken for her extraordinary support the whole time, for helpful inputs and discussions, for taking care of every nasty question and request from my side and pushing this book to success...

I hope you will enjoy the book and learn something about an exciting topic – embedded system development! And don't forget: understand the basics and practice, practice, practice to reach your targets and develop your own embedded application!

1. INTERNET OF THINGS & INDUSTRY 4.0

Internet of Things and Industry 4.0 – two buzzwords for the emerging digitalization and connection of all kinds of systems. Precise definitions for both phrases are hard to get, but nevertheless everyone has some kind of understanding for these words. Let’s try to get a common understanding for this book starting with IoT.

For most of the time communication was a pure human to human interaction, either directly or via fixed and mobile telephony. When the internet evolved during the 70's and 80's things began to change. In the first step, the internet provided a smart network for any kind of data or content exchange and for communication like email. Adding more complex services to the World Wide Web the internet entered the “Web2.0” phase providing services like e-commerce or e-productivity. With the emerging availability of internet-capable mobile devices and in particular apps for these devices, the internet evolved towards an “Internet of people”. Social media like Skype, Facebook or Instagram enable totally new ways for communication between people. This is where we are today – and now the internet extends towards smart devices and objects, replacing people to generate the data and content – we finally arrived at the “Internet of Things” or IoT. The communication takes place between machines and devices, e.g. for tracking, identification, monitoring, metering, ... Just make devices smart, connect everything with everything and connect it to the internet. As a consequence of this trend, the number of connected devices – things for the Internet of Things – explodes. Within the last years the number of things within IoT exceeds the number of people connected to the internet by far reaching some 10 billion devices in 2017. The Internet turns from the Internet of People to the Internet of Things. And all these smart devices generate a huge amount of data in a decentralized manner. This big data enables new applications, functionalities and business models – as long as you are able to analyse the data and extract valuable information from the big data.

Let’s look on the other hand at the history of industrialization: the first step was the mechanization of work using water power or steam power – “Industry 1.0”. This step increased the efficiency and productivity drastically. The next step was the mass production in well-structured assembly lines instead of individual hand crafting – “Industry 2.0”. This mass production was based on the division of labor as introduced by Henry Ford for the production of the Ford Model T. This new method reduced the production cost drastically. The common use of computer technology and automation systems enabled the third step of industry – “Industry 3.0”. The amount of manual work was reduced or even vanished and the work was done by automatic machines. Again this step enhanced the productivity drastically and further reduced production costs. This automation is standard in industry as of today, and according to definitions like the DIN 19233 standard “Automation enables equipment or systems to work (partly) without human intervention”. As the “partly” indicates, the degree of automation is flexible and depends on the degree of autonomous operation of the system. Targets for the automation are manifold:

- Save labor
- Save energy
- Reduce cost
- Increase productivity
- Improve quality and reliability
- Safer operation

Automation systems in industrial applications are very hierarchically structured, e.g. like an automation pyramid (Fig. 1). Functionality is clustered in levels with dedicated connections within each level and to adjacent levels. The sensors and actuator act on the lowermost process level with low complexity but high real-time requirements. Going up in hierarchy the complexity increases and the real-time requirements decrease until the company management level. Today, automation systems can be found nearly everywhere, not only in industrial systems: cars, white goods, home, energy systems, ...

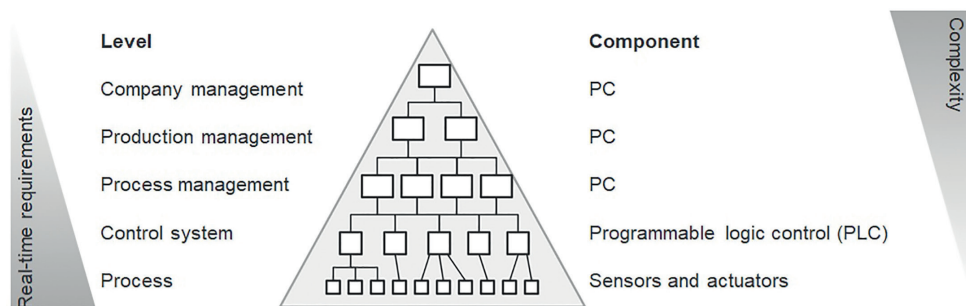


Fig. 1: Automation pyramid

In the next step, Industry 4.0 describes the current trend in industrial automation systems towards a modular structured smart factory: break the hierarchical structure and use smart devices, IoT and smart software and algorithms. In fact, the term “Industry 4.0” was created by the German government: combine smart devices and machines with communication, powerful software and algorithms to form so called “Cyber Physical Systems” (CPS) for self-organized production systems (Fig. 2). Due to this combination of connectivity, internet and algorithms the systems can act more or less autonomously – without or at least with reduced human intervention. As the data and information are available online these CPS offer many interesting opportunities and new business models, such as decentralized production, improved logistics, augmented operators or predictive maintenance.

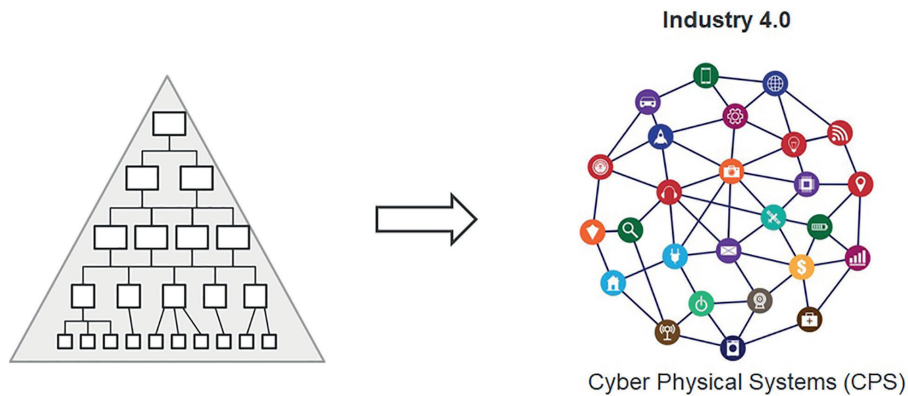


Fig. 2: From today to IoT, I4.0 and CPS

What is needed to realize IoT and Industry 4.0 devices and applications? Well – it depends... On the application, the requirements, the system, the environment, ... All these topics are widely spread, so no general answer is possible. But we can figure out some common parts that are mandatory – and some additional components, requirements and features.

From the hardware point of view we need sensors to measure and observe our system. Then we need smart hardware to perform the calculations and algorithms for the application. Here the performance of the hardware and hence the used controller or processor mainly depends on the requirements of the application. To act again on our system actuators are needed, driven by the smart hardware. Of course we have to connect these devices by suitable communication lines using a communication infrastructure like internet. These components are more or less mandatory for IoT and Industry 4.0 applications. And, last but not least, we need software and algorithms to run the devices and the applications respectively.

Some non-mandatory features are a graphical user interface (GUI), safety and security features, reliability or real-time capability. As there is almost no hierarchy any more devices have to provide real-time capability in general (details see chapter 7). Also a simple integration into the existing system or short development times might be a challenge for smart devices.

So the key component for IoT and Industry 4.0 applications is any kind of smart hardware or devices – devices with some basic intelligence at least and with dedicated communication interfaces, not just for internet connectivity but for any suitable communication interface like ZigBee or Bluetooth. This device can be a sensor, actuator, gateway, electronic control unit (ECU) or anything needed (an example for a smart sensor is given below). Technically the smart hardware is realized in form of embedded systems in general.

1.1. EMBEDDED SYSTEMS

Everyone knows and uses general purpose computers like PCs and laptops. These devices use powerful microprocessors like Intel Core i7 or AMD Ryzen 7 to realize general purpose functionality to the user – e.g. office applications, multimedia or gaming. A key parameter for the user is the performance to provide maximum computational power. Nevertheless, these microprocessors are just a minority compared to processors and controllers used in embedded systems.

Embedded systems are controller/computer systems with dedicated functionality, e.g. for control or monitoring applications. In general they are part of a larger (mechatronic) system. In Fig. 3 a typical embedded systems with sensors and actuators is depicted: sensors provide their data to the embedded hardware. The required functions are realized in software using the underlying hardware, e.g. a microcontroller. Besides driving the actuators, the embedded system is also connected to other systems via dedicated interfaces for communication and data exchange.

As the name implies, the controllers in embedded systems are embedded into the system and are invisible to the outside world. In best case the user does not even know there is a controller running. Imagine a modern car like a Mercedes S-class – some hundred embedded systems are running inside the car to realize

systems like anti-lock braking system (ABS), motor control or highly sophisticated Advanced Driver Assistance Systems (ADAS). And as long as everything is running fine the driver does not realize that there are so many embedded systems running (okay, if there is any malfunction it might getting difficult...). Besides in automotive applications embedded systems are found almost everywhere – in consumer, industrial, commercial and medical applications.

As embedded systems are in general part of a larger system they are restricted by many constraints:

- Available space/small size
- Energy consumption
- Cost
- Lifetime
- Reliability
- Safety requirements
- Environmental conditions

To meet the restrictions and find the best fitting solution embedded systems are dedicated to a specific functionality and application. One consequence of the restrictions is the limitation of processing power, which makes programming and interaction significantly more difficult compared to general-purpose computers. Besides the restrictions there are many other challenges for the development of embedded systems, in particular in case the embedded system (or device) is part of a distributed system with many devices. These challenges cover technical as well as non-technical aspects.

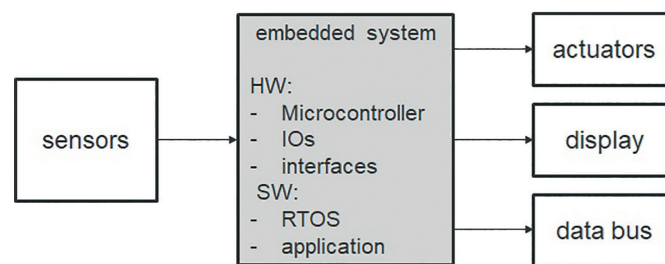


Fig. 3: Schematic of an embedded system

Starting at the device level already the basic smart hardware, e.g. the microcontroller, is a very complex system – just read more than thousand pages of the data sheet of modern microcontrollers like the Renesas Synergy™ Platform. On top of the hardware resides the software, which in turn is complex again, with low level drivers for direct hardware access and top level application software. As many systems require real-time capability, e.g. for motion control applications, both hard- and software have to enable the deterministic timing behaviour. Therefore some additional software, a real-time operating system, might be necessary. And both hard- and software have to be tested and validated intensively. And testing is getting even more difficult for connected devices and distributed systems.

Next step is to connect the device to other devices and systems – either to build a distributed system or just for data transfer. As there are dozens of different communication technologies, it starts with the definition of the most suitable interface. Is it for short or long range communication, to just one device or maybe to the internet, wireless or tethered, low or high data rate, dedicated safety requirements, reliability, ... After the communication is established, what about the security to prevent any unauthorized access to the device and system? In case the embedded device is part of a distributed system the situation is even getting worse as the complexity again increases. A next level of application software is added and maybe even artificial intelligence is used to realize the application.

All the different tasks like described above have to be executed and need the fitting expertise – from low level hardware to complex application. So, for sure cross-functional teams are needed for the development of embedded systems and applications – experts for your application, hard- and software, embedded software, safety and security, your market and requirements, ... Hence, it is desired to reduce the complexity in development of embedded systems as much as possible. This reduction in complexity is exactly the basic idea from Renesas Synergy™. Focus on your core competencies and your targets and get everything you need from other experts. In best case just from one expert like Renesas.

In case of Renesas Synergy™ focus on your application and your market and vendors. Split your system into an application part and an "embedded" part. And get the "embedded" part – starting hardware, low level software and middleware, real-time operating system, tools – from just one supplier. One contact, one supplier to reduce your development time, cost and time to market.

But now let's have a short look at the main components of embedded systems, sensors, actuators and the central processing unit.

1.2. SENSORS & ACTUATORS

Like all creatures, technical systems also have to observe, measure and process relevant parameters of themselves or the environment. Humans use their five senses, sight, hearing, touch, smell and taste, for cognition of some optical, chemical, physical or mechanical parameter (Fig. 4). Other parameters like magnetic fields or radioactivity are imperceptible for humans as the corresponding sense is missing.

Like human senses technical sensors measure dedicated technical and physical parameters of a technical process. In a simple sensor the input parameter is converted into a primary electrical output value using suitable physical or chemical effects. The primary electrical output can be any kind of electrical signal like an analog voltage or current and is connected to the main hardware of the embedded system.

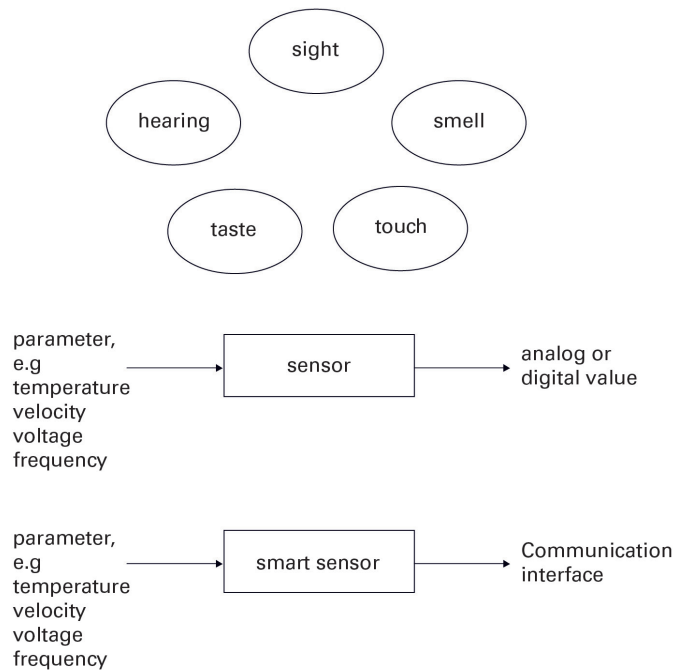


Fig. 4: Human and technical sensors (simple and smart sensor)

The functionality of simple sensors can be extended by adding additional components to make the sensor a smart sensor. E.g. the primary output can be digitized using evaluation electronics and a microcontroller. Now some algorithms can be executed already within the sensor and the digital data can be transmitted via a communication interface. In case the communication interface is a real data bus system (like Ethernet) the data can be sent to other devices and control units as well. So the sensor is a smart device itself and may be part of the IoT itself.

As space constraints are often one of the most challenging requirements for embedded systems, a key development trend for sensors is miniaturization. The size of many sensors can be shrunk by using MEMS technology (Micro-Electro-Mechanical-Systems). This technology uses semiconductor processes to produce mechanical structures in sub- μm scale together with electrical semiconductor parts to realize very small complex sensor elements.

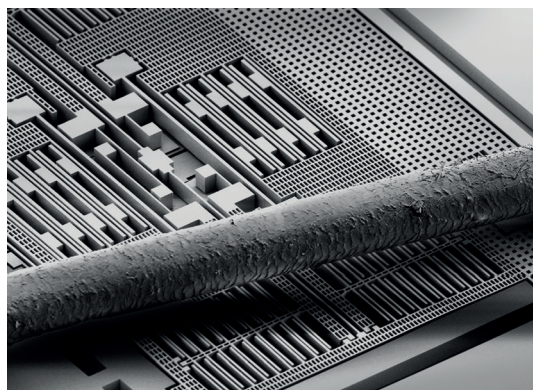


Fig.5: MEMS sensor structure compared to the size of a human hair

Actuators are used to act on the technical process to drive and control the system or process. They receive low power control signals from the smart hardware and convert these signals into driving forces, e.g. electric or hydraulic. For this conversion actuators need an energy source. Again, like for the sensors, actuators often include some smart hardware to make it a smart device itself. In conjunction with a communication interface it may then be part of the IoT itself.

1.3. CENTRAL PROCESSING UNIT & SOFTWARE

The main smart hardware of embedded systems is the processing unit. It mainly determines the performance of the system and it has to fit to the requirements of the application, like interfaces, memory, package, power consumption or price. The processing unit is programmable and runs the software. There are many different devices that can be used as a processing unit:

Of course ordinary general-purpose microprocessors can be used for complex and performant systems. In this case external chips are used for memory and peripheral modules like interfaces and data converters. For very special applications like complex signal processing applications devices like Digital Signal Controllers (DSC), Digital Signal Processors (DSP) or Field Programmable Gate Arrays (FPGA) provide the performance needed.

Despite the high performance of these devices modern embedded systems are very often based on microcontrollers. A microcontroller is a System on Chip itself as it combines a CPU core (Central Processing Unit) with dedicated peripheral modules and embedded memory for program and data. As we will see later in chapter 2 there is a large variety of microcontrollers, from small and slow 8-bit controllers to higher performance 32- and 64-bit microcontrollers. Mainly used in embedded systems nowadays are 32-bit microcontrollers. Even though the computational power of microcontrollers is limited and lower than the performance of microprocessors or DSP these devices provide many benefits for embedded systems. Therefore, microcontrollers will be discussed in detail in chapter 2.

Even the smartest hardware is useless without powerful and efficient software to realize the applications. In fact, the importance and complexity of software for the application is getting higher and higher. And so the efforts spend to develop, maintain and test the software are increasing more and more. The programming of the hardware is usually done in C/C++ providing direct access to the hardware modules and functions by low level driver. On the other hand application development is today often model-based. Here the design of complex control, signal processing or communication systems in industrial, automotive or motion control applications starts with a mathematical model of the system, e.g. using Matlab®/Simulink®. After successful development and testing of the model it has to be transferred to the target hardware. For this step often automatic code generation of C code is used. The challenge for development engineers of embedded systems is in this context, to cover both the application side and the hardware part. Rather challenging to be an application expert and an embedded hardware expert. To simplify the development and the challenges it is therefore desired to reduce the amount of C programming and to hide the complexity of the hardware. This abstraction from the hardware can be achieved by programming on a higher level of abstraction using an application programming interface (API) – see chapter 6.* /

2. MICROCONTROLLER

Microcontrollers are microprocessors with integrated hardware peripheral modules (Fig. 6). In general, the additional peripheral modules are integrated on the same chip like the CPU and provide additional functionality.

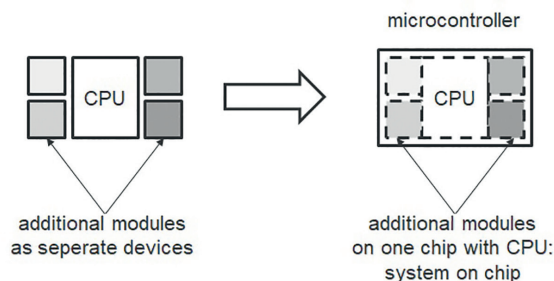


Fig.6: From microprocessor with external components (left) to an integrated microcontroller

Fig. 8 shows a microcontroller as a central component of a heating control system. Besides the CPU it uses some peripheral modules like an Analog-Digital Converter (ADC) or a LCD module to interact with the external components. Communication is done via bus interfaces and the microcontroller provides corresponding bus modules. Analog sensors like a temperature sensor can be connected to the ADC inputs and digital in- or outputs to the port module. The microcontroller also contains system function modules, e.g. for clock generation or interrupt and reset handling. This simple example already demonstrates that the needed peripheral modules are determined by the application.

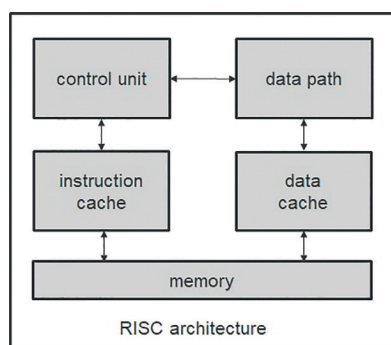


Fig.7: RISC architecture

The integrated memories are used to store dynamic data like measurement data in the RAM or static data like program code or parameter data in the flash memory. No external memory is needed – if the memory sizes are sufficient. For microcontrollers RAM sizes are in kB up to MB range and flash memory size in MB range. So do not compare these memory sizes to the memory size in GB or even TB range you know from a PC... Table 1 lists common peripheral modules and memories of microcontrollers. Important modules are also described a bit more in detail.

According to the high number of peripheral modules, there is a huge variety of microcontrollers that can be designed using the same CPU but different peripheral modules (Fig. 9). These microcontrollers differ in number and type of modules, pin count, package but use the same CPU. As the CPU remains the same the reuse of software is very high when changing from one microcontroller to another one of the same family. This kind of family concept of microcontrollers offers a high degree of flexibility and scalability to find the optimal solution for an application with respect to technical and commercial requirements.

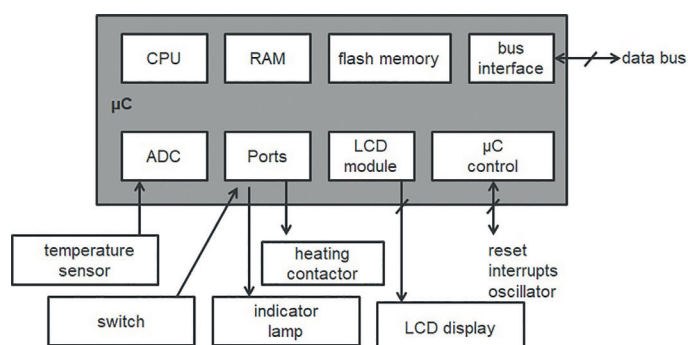


Fig.8: Peripherals of a microcontroller for a simple heating control system

As there is a huge variety of microcontrollers and microcontroller families – how to find the best fitting microcontroller for an application? First of all, the application determines the functional and environmental requirements the microcontroller has to provide, i.e. peripheral modules, performance and clock speed, package type and pin count, power consumption (both in on and off state). In addition, the application area is important, as automotive, industrial, consumer applications all have totally different requirements with regard to reliability, temperature range and environmental conditions. And of course the software requirements are important as the code and data size determine the required memory size, for volatile as well as non-volatile memory.

CPU	Central Processing Unit, 8-/16-/32-/64- bit core; multi-core possible for higher performance or redundancy
RAM	Random Access Memory, volatile and reversible memory for data
ROM	Read Only Memory, non-volatile, irreversible memory for start-up code or application code; part of chip production at semiconductor company
Flash	Non-volatile, reversible memory for program and data
Bus interface	Hardware module for protocol layer of interfaces like CAN
ADC	Analog-Digital-Converter, conversion of analog input voltages into digital values
DAC	Digital-Analog-Converter, conversion of digital values to analog output signals
Timer module	Module for counter, time measurement, pulse generation and pulse width modulation for time-dependent signals
DMA	Direct Memory Access, enables direct data transfer between modules without CPU usage
Digital ports	In- and outputs for digital signals with programmable characteristics (e.g. open-drain or push-pull output)
Analog ports	In- and outputs for analog signals, internally connected to corresponding ADC or DAC modules
Clock generation unit	Generation of system clocks, e.g. for the CPU or peripherals; contains in general several internal and external clock sources like oscillators or a PLL (phase-locked loop)
Power supply	Internal voltage regulator to generate system voltages from one voltage input (e.g. 5 V external voltage, internal generation of 3.3 V and 1.8 V)
Voltage control	Supervision of power supply with over-/under voltage detection
Watchdog timer	Timer module that is used to detect and recover from malfunctions like an infinite software loop (hang-up)

Table 1: Modules and memories of microcontrollers

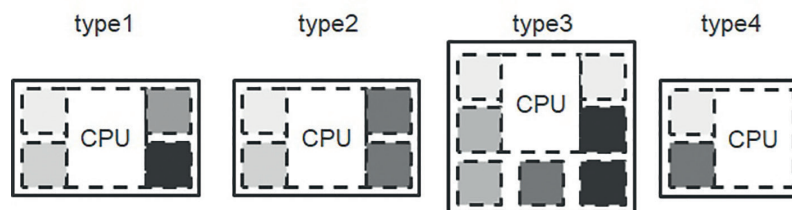


Fig.9: Family concept: different microcontrollers using the same CPU but different peripheral modules

2.1. PERIPHERAL MODULES

Some of the modules listed in Table 1 should be explained a bit more in detail as they are very common and important for embedded systems.

The connection from the IC to the outside world is handled by the I/O ports, either analog or digital. The ports can be used as general purpose I/O pins (GPIO) or they are used for alternate functions of the peripheral modules, e.g. the TxD and RxD pins of the CAN module. For most ports the port logic is configurable to switch the port functions to input, output or in/out. The port characteristics are configurable as well. In output mode the signals are driven by the port and the driving characteristics can be changed, e.g. to open drain or push-pull characteristics. As the driving capability of each port and of all ports in total is limited each port can only drive some maximum current. In input mode the input signals are captured, again with configurable characteristics.

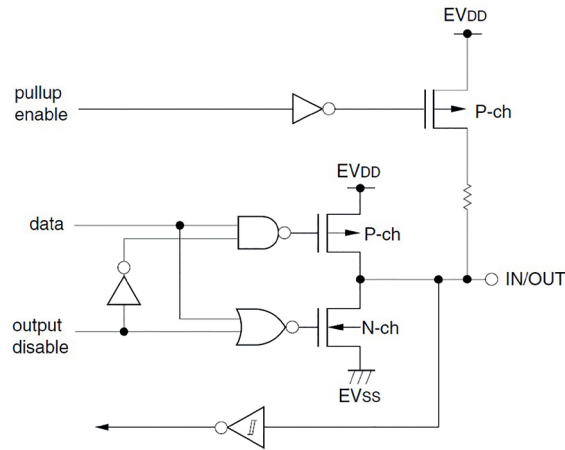


Fig.10: Schematic of a inout in/out port: flexible output configuration and Schmitt-trigger input

Communication is a major task for applications to exchange data between different components and systems. Connectivity and communication interfaces are the base for all kind of data exchange with other systems. In general, they are implementations of the data link layer of the OSI model (Open System Interconnection model) - not the physical layer. These modules take care of all the protocol handling of the corresponding bus system without CPU usage. The connection to the outside world is as alternative functions at the GPIO pins of the microcontroller. For some simple bus systems like SPI (Serial Peripheral Interface) or I²C (Inter-Integrated Circuit bus) the GPIOs represent the physical layer of the bus system. But for more complex bus systems the physical layer is realized in dedicated IC. The transceiver (combination of transmitter and receiver) communicates via digital signals with the microcontroller and converts the digital signals into the physical representation on the communication line and vice versa. Fig. 11 depicts a schematic of a CAN bus (Controller Area Network) as an example. The CAN controller modules of the two MCUs (Microcontroller Unit) send digital data via TxD pin of the GPIOs to the CAN transceiver. The transceiver converts the digital signals into the differential signal of the physical layer of the CAN bus and transmits the differential data via the twisted pair cable. Each transceiver receives these CAN data, converts the data into digital representations and sends the data via the RxD pin to the MCU.

Using hardware modules like the CAN module makes communication rather simple as the complete protocol handling is done autonomously by the module – after proper initialization of the module. More complex bus systems like Ethernet use also higher layers of the OSI model and communication is much more complex, even though many functions are again realized in hardware. Nevertheless, a sophisticated software is needed for proper Ethernet communication. To simplify the use of Ethernet (and to make you focus on your application, not the basic communication) middleware software is used which provides the desired Ethernet functionality – for details please refer to chapters 9 and 10.

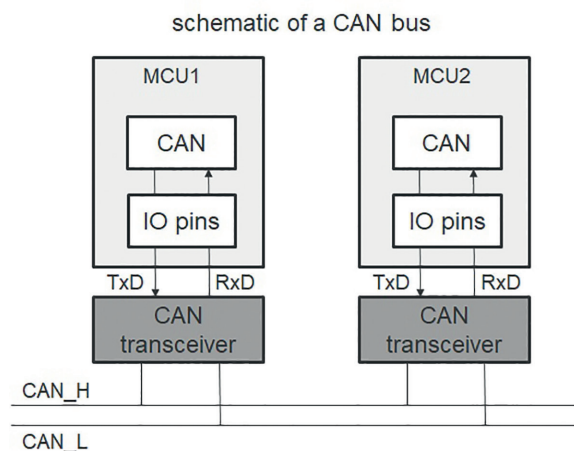


Fig.11: Schematic of a inout in/out port: flexible output configuration and Schmitt-trigger input

Timer modules consist of a set of timers (as the name implies...) and offer a high degree of flexibility to generate and measure signals and timings. In capture mode the timers capture input signals to measure interval times, periods or pulse widths. They can also be used to count external events or to generate events in compare mode. Pulses or pulse width modulation signals (PWM) can be generated in output mode, possibly using several timers.

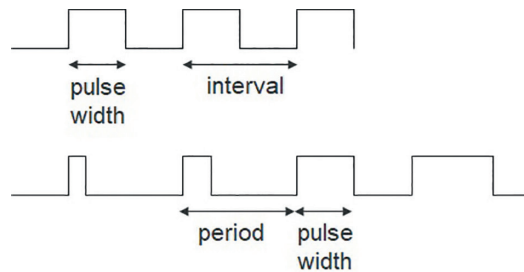


Fig.12: Timer module: capture of signals (top) and PWM generation (bottom)

Connection to the analog outside world is achieved by using ADC and DAC modules and corresponding analog input and output pins.

The ADC samples an analog input signal (continuous in time) making it a time-discrete signal using a sample-and-hold input circuit (Fig. 13). If a continuous signal is sampled the Shannon-Nyquist sampling theorem has to be fulfilled to avoid aliasing: the sampling frequency of the ADC has to be at least double or the highest frequency of the input signal. After sampling the signal – discrete in time - is quantized to its digital representation introducing a small quantization error. The number of discrete digital values depends on the resolution of the converter, common for integrated ADC are 8-, 10- or 12- bit resolution. The quantization error is also determined by the resolution of the converter. There are many different architectures available for ADCs, like SAR (Successive Approximation) or sigma-delta ADCs. The most popular architecture for integrated ADCs is the SAR ADC as it combines high performance with low power consumption.

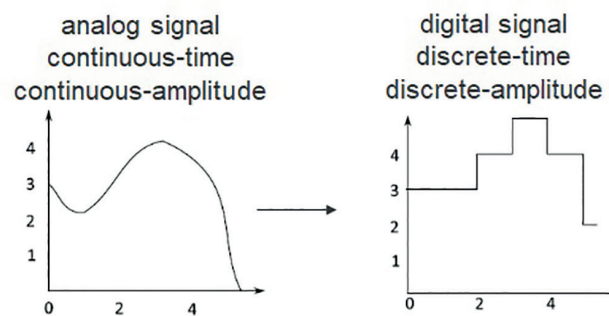


Fig.13: ADC conversion

The DAC performs the reverse function like a ADC. It converts a digital signal into an analog output signal. Again it is characterized by resolution, sampling frequency and accuracy.

All time-sensitive systems like real-time applications need a mechanism to react very fast on internal or external events. This mechanism is realized by interrupts. Interrupts do exactly what the name implies: they interrupt the running program execution. An interrupt is associated to an event – like an external input signal or a hardware signal - that needs immediate attention in case of occurrence. As soon as the event happens, the current execution is stopped and the status of the processor saved. Afterwards, the corresponding interrupt service routine (ISR) is executed to run the actions needed to serve this interrupt. After the ISR is finished, the processor continues program execution. In general, there are several interrupt sources (events) and hence several interrupts. To allow a structured handling the interrupts can be prioritized or masked. An interrupt with a lower priority cannot interrupt an interrupt with a higher priority but is executed after the high priority interrupt has finished.

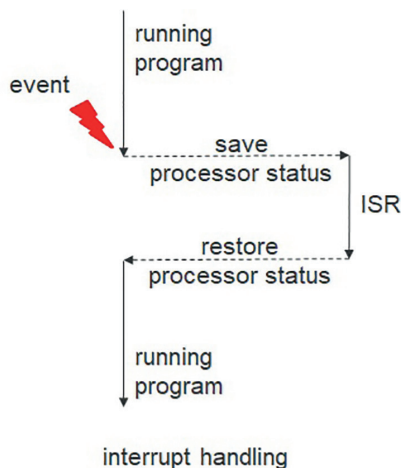


Fig.14: Basic interrupt handling

2.2. ARM ARCHITECTURES

The CPU of modern microcontrollers has basically a RISC architecture (Reduced Instruction Set Computer) as this architecture combines a high performance with a high power efficiency. Each instruction of the complete instruction set is realized by dedicated hardware within the CPU. Just simple instructions of fixed length are available. An instruction pipeline enables the parallel execution of instructions with one step delay. Due to this, pipelining instructions are executed within one clock cycle (at least mostly). Memory access is only possible for dedicated load/store instructions (load/store architecture).

A very commonly used processor architecture for microcontrollers for embedded systems is the famous ARM® architecture developed by Acorn in the early 1980's. Since then this architecture is very popular (more than 100 billion ARM® processors sold as of 2017) and is licensed by many, many companies – maybe all semiconductor companies use it in some of their microcontrollers. Of course, the core was refined over the years and currently versions ARMv7 and ARMv8 are used.

Many features of the ARM® RISC architecture support the requirements from microcontrollers and embedded systems. It combines a low power consumption with a high performance (at least for microcontroller, don't compare to microprocessors. . .) and enables a high level of integration to build a System-on-Chip (SoC). It has a Harvard architecture with separate memory and signal paths for instructions and data. A 32-bit address and data bus is supported up to ARMv7, ARMv8 supports also 64-bit. The embedded interrupt controller provides very short interrupt latencies, in particular for all kinds of real-time applications. The ARM® core has seven operating modes as listed in Table 2. The program code runs in user mode until an exception occurs that calls an exception mode (all modes but user and system mode).

Mode	Description
User	unprivileged mode for most tasks
System	privileged mode with same registers as user mode
FIQ	mode for high priority (fast) interrupts
IRQ	mode for low priority (normal) interrupts
Supervisor	entered on reset and when SW interrupt is executed
Abort	handles memory access violations
Undef	handles undefined instructions

Table 2: Operating modes of the ARM® core

The register set of the ARM® core consists of general purpose registers and special function registers and is the same for different ARM® versions. The accessibility of the registers differs depending on the operating mode. Each mode can access the general purpose registers r0 – r12, the stack pointer (r13) and the link register (r14), the program counter (r15) and the current program status register (cpsr).

Register	Number
General purpose register	30
Saved program status register	5
Current program status register	1
Program counter	1

Table 3: Registers of the ARM® core

One famous example for the ARM® core is the ARMv7E-M, also known as ARM® Cortex-M4F. It is used in Renesas Synergy series of MCUs, S3, S5 and S7 (whereas S1 uses ARM® Cortex-M0+). The popularity for embedded systems is based on a low transistor count (and hence small silicon die size) and the predictability of operation. It uses a 3 stage instruction pipeline and provides two instruction sets, the standard 32-bit ARM® instruction set and a 16-bit Thumb and Thumb2 instruction set. Dedicated DSP features support many kinds of digital signal processing applications and an embedded floating point units (FPU, IEEE-754 compliant) enables floating point operations. Up to 240 interrupts are available and the interrupt latency is very short, just 12 clock cycles.

2.3. RENESAS SYNERGY™ MCU FAMILYS

After the short general introduction to microcontrollers and the ARM® architecture let's move towards the Renesas Synergy™ MCUs. As depicted in Fig. 15, the family contains four series, S1, S3, S5 and S7. The target of the Sx family is to provide maximum flexibility for any kind of application keeping the maximum compatibility between the single devices. Each series targets at a dedicated application area, from low power applications like mobile and battery based systems for the S1 series to high performance control and display applications for the S7 series.

The four series share some common features like the silicon process. The S1 and S3 series use a 130 nm silicon process which is optimized for low power operation. The supply voltages can be from 1.6 V up to 5.5 V. On the other hand the S5 and S7 series use a high performance 40 nm silicon process and operate on 2.7 V to 3.6 V. The operating temperature range for all device is -40 °C – 105 °C.

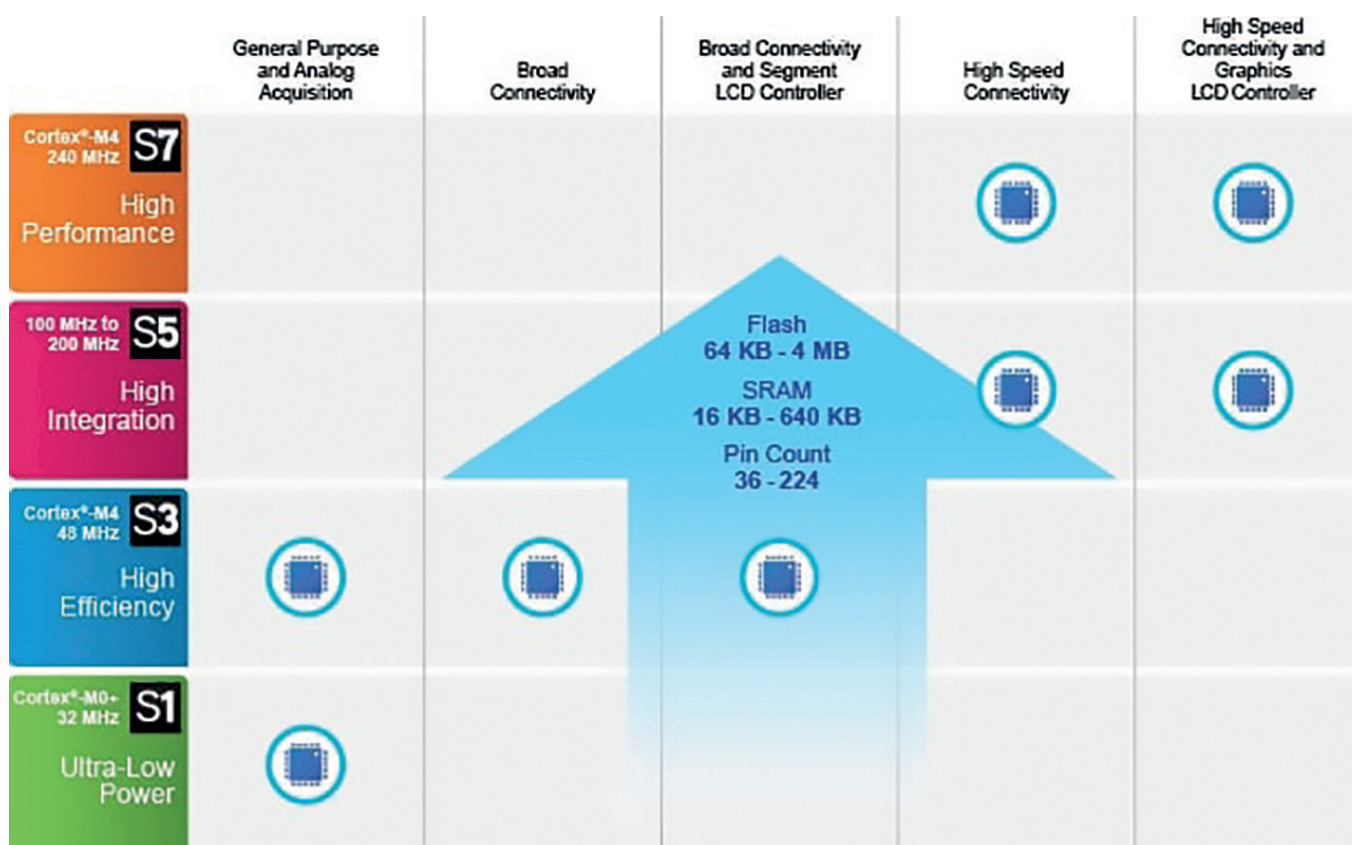


Fig.15: Schematic of Renesas Synergy™ MCUs

One key aspect of MCU families is the flexibility and compatibility of the devices of the series. The very fundamental part is the CPU that is the same for all devices of a series (and very similar within the family like the ARM® Cortex-M0+ for S1 series and ARM® Cortex-M4F for the other three series). This usage of the same CPU enables the reuse of software when changing the device – a great benefit for the development of new applications or for changing requirements during a development. But also other features can provide a high degree of flexibility and compatibility and were taken into account during the development of the Synergy series. Some of these features like memory size and pin count are shown in Fig. 15 and Fig. 16.

Each series is available in different packages and with different pin counts. E.g. the pin count ranges from 100 to 224 for the S7 series and they use package types like BGA, LQFP or LGA. This variety of packages and pin count offers a drop-in pin-to-pin compatibility within a given MCU series and the selection of the best fitting device for an application – keeping the same CPU! So changing from a 100 pin LQFP device to a 224 pin BGA device is rather simple and straight forward – at least with regard to the software, of course you need to change your board and PCB...

The embedded memory also provides a wide scalability, both for the RAM and the flash memory. Up to 4 MB flash memory is available for the S7 series – or just 64 kB. As the memory size is an important factor for the die size (and hence the price...) it can be adapted easily to the requirements of the application. Scalability and compatibility were also taken into account for the peripheral modules. Each peripheral has the same baseline functions in all MCUs. For higher MCUs the features of the peripherals scale upward in function – again giving a great benefit for code reuse.

The four series of the Synergy family will shortly be introduced in the following sections to give a first overview. The S7 series will be explained a bit more in detail as one device out of this series is used in the Starter Kit that builds the basic hardware for the lab of this book.

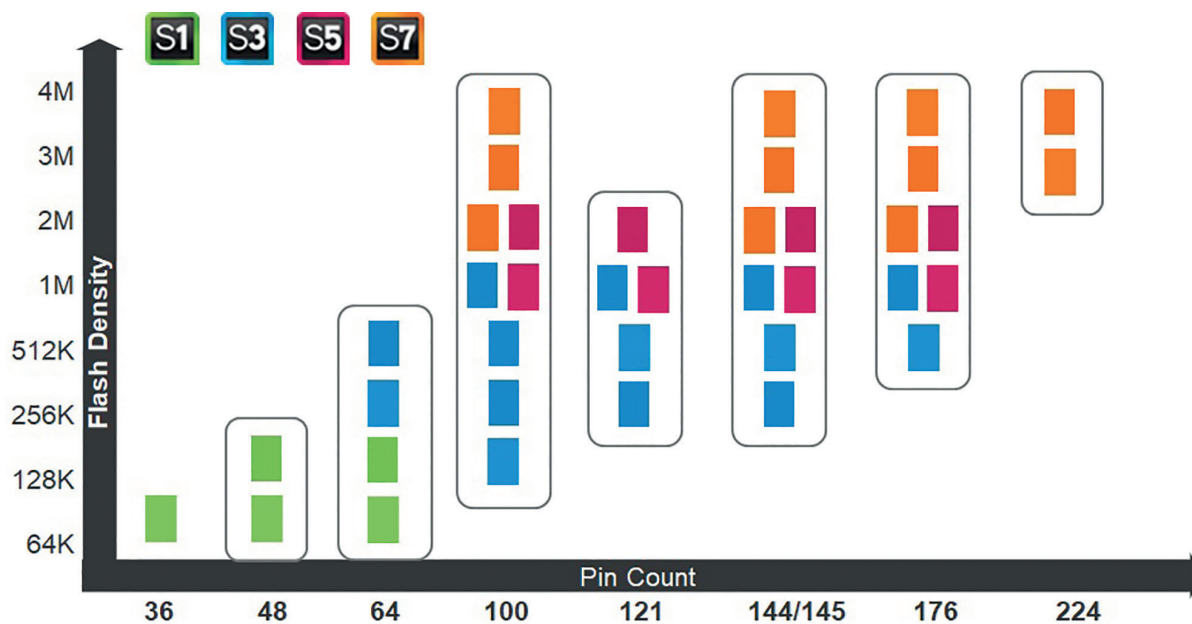


Fig.16: Portfolio of Sx family as of 2017

S1 SERIES

The S1 series of the Renesas Synergy™ MCUs is the power-efficient series using the ARM® Cortex-M0+ CPU. Low voltage operation down to 1.6 V is possible due to the dedicated silicon technology. The power consumption is as low as 70.3 μ A/MHz in operating mode. Even at the maximum clock speed of 32 MHz the maximum power consumption is just about 2.3 mA. To reduce the power consumption even more dedicated low power modes are implemented. In software standby mode the power consumption drops down to 440 nA. Three package types with up to 64 pins are available for the 9 devices of the S1 series (as of 2017). Besides a 16 kB SRAM and 4 kB data flash, the code flash can be 128 kB and 256 kB. Fig. 17 shows the features and peripherals for the S1 series of MCUs. Connectivity is realized by corresponding interface modules and covers short range connections like I²C and SI as well as standard bus systems CAN and USB. Besides analog and timer modules, the devices incorporate special function modules as well. A dedicated safety module offers safety functionality for both hardware and software like memory protection, ADC diagnostics or a watchdog timer (IWDT). A capacitive touch sensor can be attached using the HMI peripheral.

The feature set in combination with the low power consumption make the S1 series the excellent choice for all kinds of mobile and battery-operated devices.

32-MHz ARM® Cortex®-M0+ CPU				S1				NVIC SWD MTB			
Memory		Analog		Timing & Control		HMI					
Code Flash (128 KB)		14-Bit A/D Converter (18 ch.)		General PWM Timer 32-bit		Capacitive Touch Sensing Unit (31 ch.)					
Data Flash (4 KB)		12-Bit D/A Converter		General PWM Timer 16-bit x6							
SRAM (16 KB)		Low-Power Analog Comparator x2		Asynchronous General Purpose Timer x2							
Security MPU		Temperature Sensor		WDT							
Connectivity		System & Power Mgmt		Safety		Security & Encryption					
USBFS		Data Transfer Controller		SRAM Parity Error Check		128-bit Unique ID					
CAN		Event Link Controller		Flash Area Protection		TRNG					
Serial Communications Interface x3		Low Power Modes		ADC Diagnostics		AES (128/256)					
SPI x2		Multiple Clocks		Clock Frequency Accuracy Measurement Circuit							
IIC x2		Port Function Select		Data Operation Circuit							
		RTC		CRC Calculator							
		SysTick		Port Output Enable for GPT							
				IWDT							

Fig. 17: Features and peripherals of the S1 series

S3 SERIES

Higher performance compared to the S1 series is provided by the S3 series. Again one target is connected portable applications, but also mid-range HMI applications and building control are in the focus area of the S3 series. To serve the requirements of these applications the series provides the fitting core (ARM® Cortex-M4 CPU with FPU), features and peripheral modules. Maximum power consumption at full speed operation of 48 MHz is about 12 mA, in software standby mode it drops down to 900 nA. 7 devices (as of 2017) are available in four package types and with up to 145 pins. The memory size is increased significantly up to 1 MB of code flash memory for more complex and powerful applications. Enhanced features of the peripheral modules support the usage for the target application. The number of interfaces is increased and additional interfaces are available, like a SDHI interface for data transfer to and from a SD card or multimedia card interface (MMC). For HMI applications an additional segment LCD controller module is available.

48-MHz ARM® Cortex®-M4 CPU				S3				FPU MPU NVIC ETM JTAG SWD Boundary Scan			
Memory		Analog		Timing & Control		HMI					
Code Flash (1 MB)		14-Bit A/D Converter (28 ch.)		General PWM Timer 32-bit x10		Capacitive Touch Sensing Unit (31 ch.)					
Data Flash (16 KB)		12-Bit D/A Converter x2		Asynchronous General Purpose Timer x2		Segment LCD Controller					
SRAM (192 KB)		Low-Power Analog Comparator x2		WDT							
Flash Cache		High-Speed Analog Comparator x2									
Security MPU		OPAMP x4									
Memory Mirror Function		Temperature Sensor									
Connectivity		System & Power Mgmt		Safety		Security & Encryption					
USBFS		DMA Controller (4 ch.)		ECC in SRAM		128-bit Unique ID					
CAN		Data Transfer Controller		SRAM Parity Error Check		TRNG					
SDHI/MMC		Event Link Controller		Flash Area Protection		AES (128/256)					
Serial Communications Interface x6		Low Power Modes		ADC Diagnostics		GHASH					
IrDA Interface		Multiple Clocks		Clock Frequency Accuracy Measurement Circuit							
QSPI		Port Function Select		Data Operation Circuit							
SPI x2		RTC		CRC Calculator							
IIC x3		SysTick		Port Output Enable for GPT							
SS1 x2				IWDT							
External Memory Bus											

Fig. 18: Features and peripherals of the S3 series

S5 SERIES

A perfect blend of performance and integration in a mid-range line of MCUs makes the Renesas Synergy™ S5 Series MCUs a selection that covers many embedded system requirements. Maximum clock speed is 120 MHz for the S5 series. Processing performance of the ARM® Cortex-M4 CPU with FPU is combined with large memory (up to 2 MB code flash), connectivity, extensive built-in security and safety features, analog modules, and economical control of low-cost colour TFT LCD panels for tackling many IoT applications. With a full connectivity suite, graphics engine, high precision data acquisition analog interfaces and an extensive number of packages, the S5 Series is targeted for use in advanced HMI and industrial applications. Three

package types for the 10 devices and pin counts up to 176 pins make up the flexibility and scalability of this series. Typical applications for the S5 series spread from motion and position control via metering and HMI applications to optical networking and communication gateways.

120-MHz ARM® Cortex®-M4 CPU		S5		FPU MPU NVIC ETM JTAG SWD Boundary Scan	
Memory <ul style="list-style-type: none"> Code Flash (2 MB) Data Flash (64 KB) SRAM (640 KB) Flash Cache Security MPU Memory Mirror Function 	Analog <ul style="list-style-type: none"> 12-Bit A/D Converter x2 (21 ch.) 12-Bit D/A Converter x2 High-Speed Analog Comparator x6 PGA x6 Temperature Sensor 	Timing & Control <ul style="list-style-type: none"> General PWM Timer 32-bit Enhanced High Resolution x4 General PWM Timer 32-bit Enhanced x4 General PWM Timer 32-bit x6 Asynchronous General Purpose Timer x2 WDT 	HMI <ul style="list-style-type: none"> Capacitive Touch Sensing Unit (18 ch.) Graphics LCD Controller 2D Drawing Engine JPEG Codec Parallel Data Capture 		
Connectivity <ul style="list-style-type: none"> Ethernet MAC Controller Ethernet DMA Controller Ethernet PTP Controller USBHS USBFS CAN x2 SDHI/MMC x2 Serial Communications Interface x10 IrDA Interface QSPI SPI x2 IIC x3 SSI x2 Sampling Rate Converter External Memory Bus 	System & Power Mgmt <ul style="list-style-type: none"> DMA Controller (8 ch.) Data Transfer Controller Event Link Controller Low Power Modes Multiple Clocks Port Function Select RTC SysTick 	Safety <ul style="list-style-type: none"> ECC in SRAM SRAM Parity Error Check Flash Area Protection ADC Diagnostics Clock Frequency Accuracy Measurement Circuit CRC Calculator Data Operation Circuit Port Output Enable for GPT IWDT 	Security & Encryption <ul style="list-style-type: none"> 128-bit Unique ID TRNG AES (128/192/256) 3DES/ARC4 RSA/DSA SHA1/SHA224/ SHA256 GHASH 		

Fig. 19: Features and peripherals of the S5 series

S7 SERIES

The Renesas Synergy™ S7 Series combines highest performance with advanced connectivity and superior security features. The large 4 MB code memory is suitable for complex application code and cancels the need for an external memory. The ARM® Cortex-M4 CPU with FPU running at 240 MHz in combination with the zero wait-state fast SRAM enables complex algorithms and high-speed calculations for any kind of application. Multiple high-speed connectivity channels with multi-layer internal busses can operate simultaneously, even simultaneously with the powerful graphics engine to drive vibrant colour TFTLCD panels. For secure communication and secure IoT devices a sophisticated security and encryption module provides extensive hardware acceleration for features like cryptography, HASH algorithms or secure key generation.

11 devices in three package types and pin counts up to 224 pins complete the features to provide a flexible and scalable series. Target applications cover complex HMI interfaces, communication infrastructure, industrial automation and PLC.

240-MHz ARM® Cortex®-M4 CPU		S7		FPU MPU NVIC ETM JTAG SWD Boundary Scan	
Memory <ul style="list-style-type: none"> Code Flash (4 MB) Data Flash (64 KB) SRAM (640 KB) Flash Cache Security MPU Memory Mirror Function 	Analog <ul style="list-style-type: none"> 12-Bit A/D Converter x2 (25 ch.) 12-Bit D/A Converter x2 High-Speed Analog Comparator x6 PGA x6 Temperature Sensor 	Timing & Control <ul style="list-style-type: none"> General PWM Timer 32-bit Enhanced High Resolution x4 General PWM Timer 32-bit Enhanced x4 General PWM Timer 32-bit x6 Asynchronous General Purpose Timer x2 WDT 	HMI <ul style="list-style-type: none"> Capacitive Touch Sensing Unit (18 ch.) Graphics LCD Controller 2D Drawing Engine JPEG Codec Parallel Data Capture 		
Connectivity <ul style="list-style-type: none"> Ethernet MAC Controller x2 Ethernet DMA Controller Ethernet PTP Controller USBHS USBFS CAN x2 SDHI/MMC x2 Serial Communications Interface x10 IrDA Interface QSPI SPI x2 IIC x3 SSI x2 Sampling Rate Converter External Memory Bus 	System & Power Mgmt <ul style="list-style-type: none"> DMA Controller (8 ch.) Data Transfer Controller Event Link Controller Low Power Modes Switching Regulator Multiple Clocks Port Function Select RTC SysTick 	Safety <ul style="list-style-type: none"> ECC in SRAM SRAM Parity Error Check Flash Area Protection ADC Diagnostics Clock Frequency Accuracy Measurement Circuit CRC Calculator Data Operation Circuit Port Output Enable for GPT IWDT 	Security & Encryption <ul style="list-style-type: none"> 128-bit Unique ID TRNG AES (128/192/256) 3DES/ARC4 RSA/DSA SHA1/SHA224/SHA256 GHASH 		

Fig. 20: Features and peripherals of the S7s series

S7G2 MCU

The S7G2 MCU in 176-pin LQFP package is one device from the S7 series and besides some ADC channels and some CTSU pins (Capacitive Touch Sensing Unit) it provides a superset of functions of the Renesas Synergy microcontrollers. This device is also part of the S7G2 starter kit that is used in the lab. To get an impression of the complexity and functionality of this sophisticated piece of silicon just read the User's Manual of the S7G2 MCU – just about 2100 pages. It is obvious that not all features of this device can be described here in detail, but some major features will be introduced in the next sections.

A first overview about the device is already given in Fig. 20. The heart of this MCU is an ARM® Cortex-M4 core with a maximum operating frequency of 240 MHz. A dedicated FPU (Floating Point Unit) supports single precision floating point operations. It supports low power modes as well as memory protection by a Memory Protection Unit (MPU).

The interrupt handling is done in two steps using a dedicated Interrupt Control Unit (ICU) in combination with the Nested Vectored Interrupt Controller (NVIC) of the ARM® core (Fig. 21). This combination provides short interrupt latency, high flexibility and configurable prioritization for up to 316 interrupt sources from peripherals and 16 external pin interrupts.

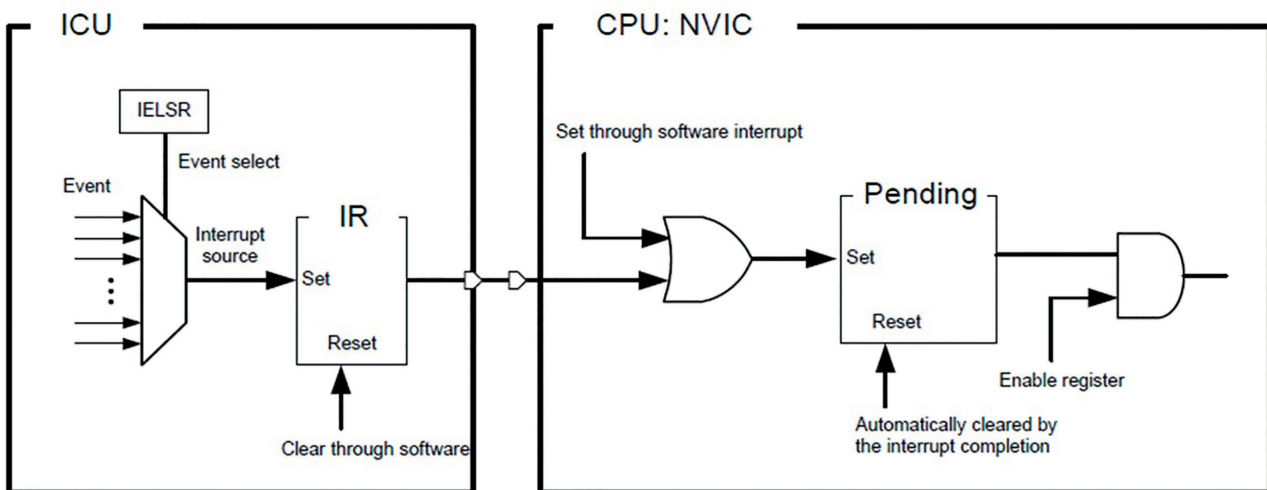


Fig. 21: Interrupt path of the ICU and NVIC

Even though the S7G2 MCU provides a 4 GB linear address space just parts of these addresses are correlated to a physical memory and in addition, the underlying memories are very different – volatile, permanent, ... A 4 MB code and a 64 kB data flash build the permanent memories of the S7G2. Also two volatile SRAMs are provided. A 640 kB high-speed SRAM for data and parameter storage is protected by a parity bit or double-bit error detection. The 8 kB Standby SRAM retains its data even during deep software standby mode.

CONNECTIVITY

As already depicted in Fig. 20 the S7G2 has a large bundle of peripheral modules to provide maximal flexibility for any kind of application. In particular, several connectivity modules enable many different ways to connect the device – a key feature for IoT applications. The set of interfaces starts with modules for local interconnection to sensors or other ICs on the same PCB. The Serial Communication Interface (SCI) can be configured to five different asynchronous and synchronous interfaces covering UART, simple I²C and SPI and a smart card interface. For more complex I²C and SPI communication dedicated peripheral modules are available. Up to 3 I²C channels can be used with a data rate of up to 1 Mbps, in master, multi-master or slave mode. The SPI interface can be configured to the needs of this non-standardized interface in a very flexible manner. E.g. with regard to data format, transfer bit length or clock polarity. Two SPI channels can run in high-speed and full-duplex mode. More complex communication interfaces provided by the S7G2 MCU are CAN, USB 2.0 and Ethernet.

CAN (Controller Area Network) is an asynchronous bus system for half-duplex data transmission up to 1 Mbps. Since 1993 it is specified in ISO11898 and covers the layers 1 and 2 of the OSI model and nowadays it is commonly used in automotive, industrial, automation and medical applications. Layer 1 is not implemented in microcontrollers in general but is realized in dedicated CAN transceivers. In contrast, layer 2 is part of many microcontrollers like the S7G2. The CAN controller of the S7G2 is fully compliant to the ISO 11898-1 standard and provides two CAN channels. It supports standard (11-bit) as well as extended (29-bit) message identifier and up to 32 mailboxes. Both CAN types, low-speed CAN (up to 125 kbps) and high-speed CAN (up to 1 Mbps), are available.

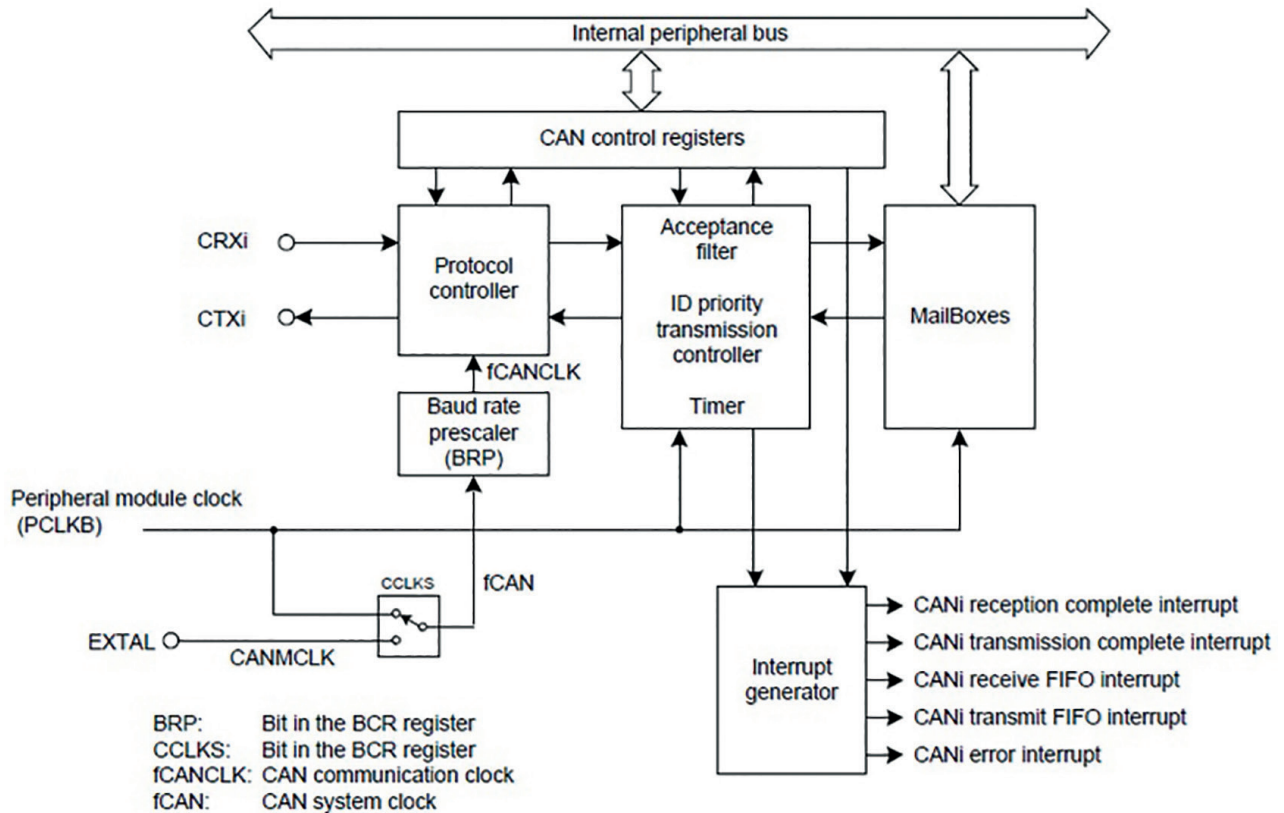


Fig. 22: CAN module block diagram

To connect external components like memory sticks or displays the famous USB interface (Universal Serial Bus) is commonly used. The S7G2 provides a USB 2.0 interface and includes both the protocol module and the transceiver. It can be used as a full-speed USB controller or as a high-speed USB controller, both in host or device mode. In full-speed mode it supports full- and low-speed transfer according to USB 2.0 specification with 12 Mbps and 1.5 Mbps respectively. In high-speed mode also high-speed transfer with 480 Mbps is possible.

Ethernet is by far the most popular and important bus system for any kind of networking between ECUs (Electronic Control Unit) and systems, e.g. for LAN (Local Area Network) or WAN (Wide Area Network). And of course it is the backbone of any IoT system. According to the Ethernet standard IEEE 802.3 layers 1 and 2 of the OSI model are specified. On top of these two layers additional layers are defined to build a complete layer stack of the OSI model. For internet application for example these additional layers are TCP/IP (Transmission Control Protocol/Internet Protocol) in layer 3 and 4 and HTTP (Hypertext Transfer Protocol) in layers 5 to 7 (Fig. 23).

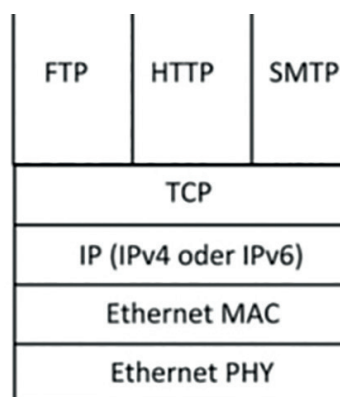


Fig. 23: TCP/IP OSI model

One of the reasons for the success of Ethernet is the strict separation of layers 1 and 2. Layer 2, the Ethernet MAC (Media Access Control), is nearly unchanged since a long time. On the other hand layer 1, the Ethernet PHY, is changing continuously to enable higher data rates of up to 400 Gbps (Fig. 24).

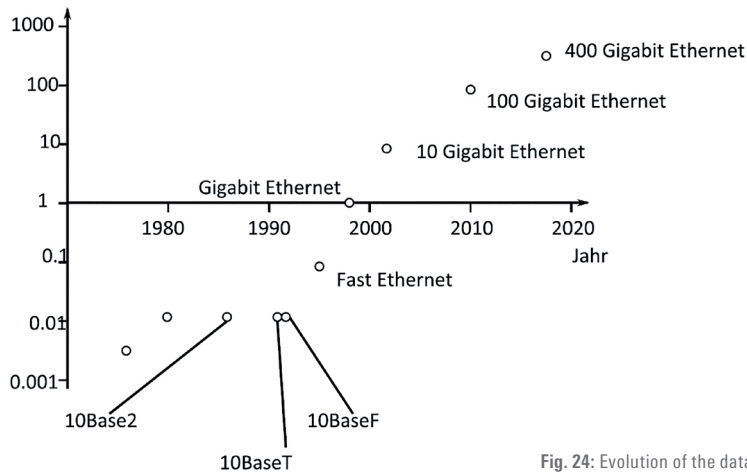


Fig. 24: Evolution of the data rate of Ethernet

The strict separation of the two layers is also reflected in the hardware realization of Ethernet. Layer 1 is realized in a dedicated Ethernet PHY transceiver whereas layer 2 is in general a hardware module of a microcontroller. The interface between the microcontroller and the transceiver is standardized and is called MII (Media Independent Interface).

The S7G2 provides full Ethernet MAC functionality for two channels compliant with IEEE 802.3 MAC layer protocol standard for data rates of 10 Mbps and 100 Mbps in half-duplex and full-duplex mode. A Precision Time Protocol module handles the timing and synchronization according to IEEE 1588-2008 and an Ethernet DMA controller enables internal data transfer without CPU load.

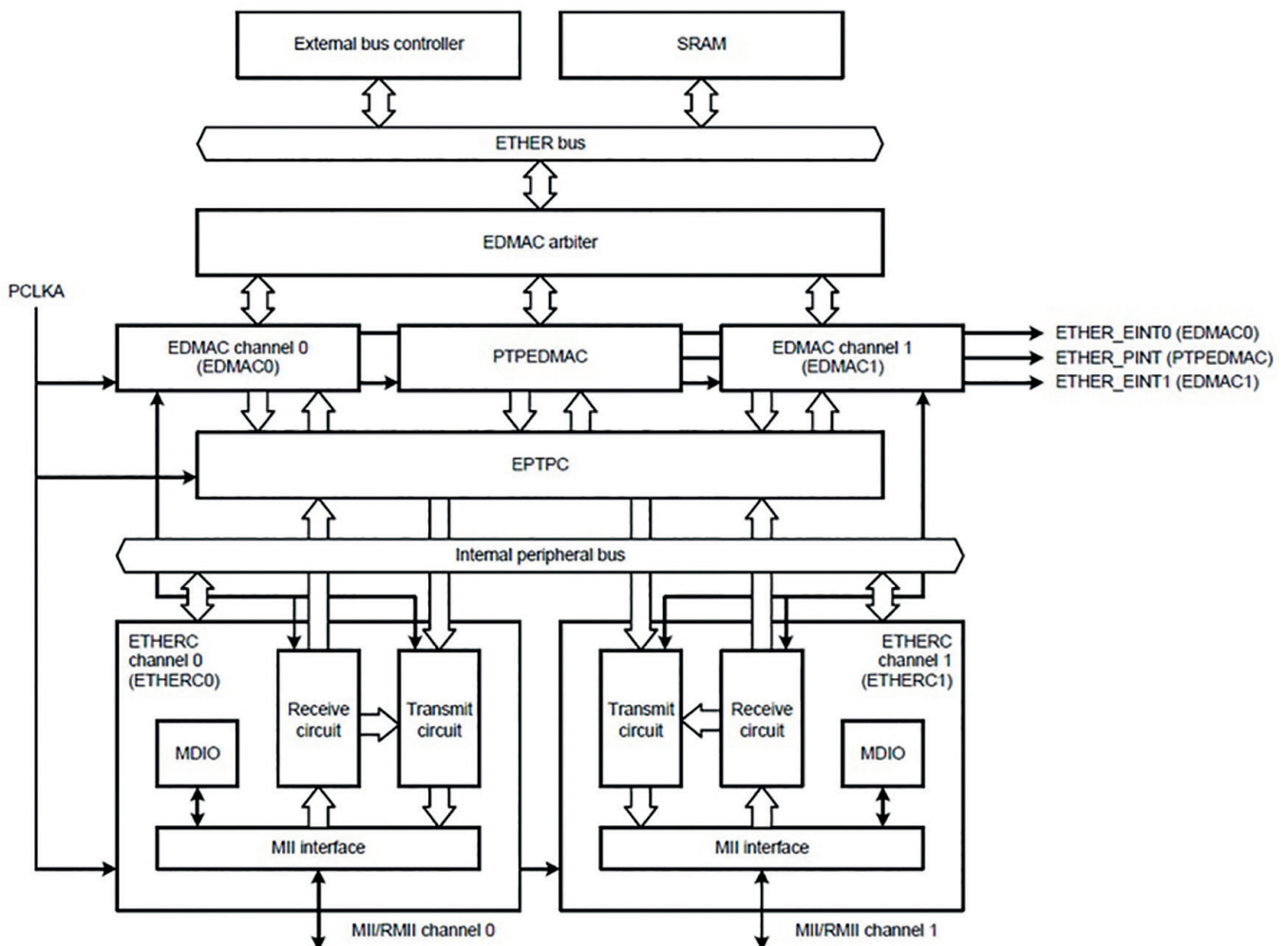


Fig. 25: Block diagram of Ethernet MAC module

SECURE CRYPTO ENGINE

Connectivity is the key aspect of IoT applications and offers many benefits for these applications. On the other hand, the connectivity makes the system vulnerable to data corruption or cyberattacks. Therefore data security and integrity is an important topic for connected systems. The protection can be done by encryption of the data. If the encryption and decryption is done by software a lot of software performance is needed, in particular for complex and sophisticated algorithms. The CPU can be relieved by dedicated hardware support. A Secure Crypto Engine (SCE 7) provides several simple-to-use standard encryption methods to perform the encryption and decryption independent of the CPU. These methods include symmetric security algorithms like ASC (Advanced Encryption Standard) or 3DES (Data Encryption Standard) as well as asymmetric security algorithms like RSA (Rivest-Shamir-Adleman cryptosystem) or DSA (Digital Signature Algorithm). In addition, it can generate random numbers using a True Random Number Generator (TRN), hash-values or 128-bit unique IDs.

TIMER

The S7G2 provides several timer modules to support and enable many timing related features like signal generation and input measurements. The 32-bit General Purpose Timer GPT32 has 14 channels and in output mode it can easily generate pulse, pulse patterns and PWM signals (Pulse Width Modulation). In input mode it can measure pulse width or capture input events. A 16-bit Asynchronous General-Purpose Timer (AGT) can be used to generate pulse, count external events or measure periods of signal. The function of realtime clock and watchdog timer are realized by dedicated hardware modules as well.

ADC & DAC

To connect analog components like analog sensors the S7G2 has both ADCs and a DAC. Two SAR ADCs (Successive Approximation) ADC12 can operate in 12-, 10- and 8-bit conversion mode for up to 21 analog input channels and combines a rather short conversion time (down to 0.4 μ s per channel) with a high resolution. The selection of the analog input to be converted can be done by different operating modes like single scan mode, continuous scan mode or group scan mode. For safety reasons the ADC implements a self-diagnosis and dedicated reference voltage input pins can be used to increase the accuracy. In addition an internal temperature sensor is connected to the ADC1.

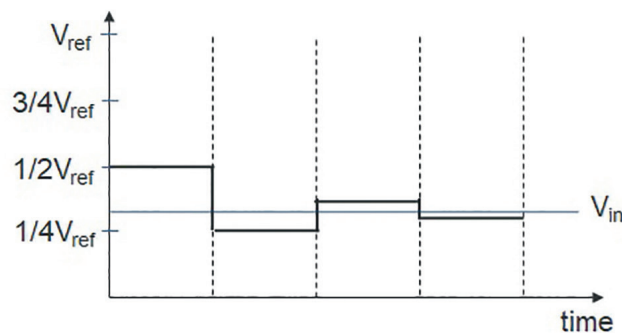


Fig. 26: Basic working principle of a SAR ADC (here 4 bit for simplicity)

Two independent output channels of the DAC12 can drive two analog outputs DA_x with 12 bit resolution according to the digital data in DADR0 and DADR1 respectively:

$$DA_x = \frac{DADR_x}{2^{12}} \cdot V_{\text{refhigh}}$$

For a capacitive load at the DA_x outputs of 20 pF the conversion time is as low as 3.0 μ s.

HMI

The interaction between humans and machines is getting more and more important, and hence the human-machine interface (HMI) plays a major role for interactive systems. Simple control elements like switches, buttons and indication lights are nowadays often insufficient to interact with the machine. More sophisticated HMIs like displays or touch displays – to provide a colorful image of important data and parameter and the touch feature to control the machine - are needed for many IoT applications to enable smooth and smart interaction with the system.

To make the integration of sophisticated HMIs and graphics as easy as possible the S7G2 provides powerful HMI and graphic modules. For example, the Capacitive Touch Sensing Unit (CTS) measures the electrostatic capacitance of a touch sensor. Changes in the electrostatic capacitance of the touch sensor are determined by software that enables the CTSU to detect whether a finger is in contact with the sensor. Up to 12 channels are available for the S7G2 and it provides several operation modes, e.g. for self-capacitance and mutual capacitance measurement.

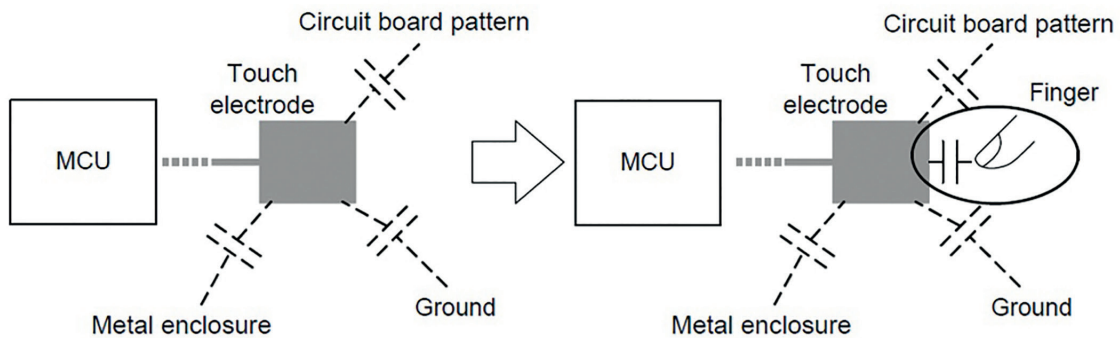


Fig. 27: Schematic of a touch sensor connected to a MCU

The connection to LCD displays is supported by a dedicated Graphics LCD Controller (GLCDC). The GLCDC can be configured in a very flexible way and it supports multiple data formats like 32- and 16-bit/pixel graphics data and 8-, 4-, and 1-bit LUT data formats. Up to three planes can be superimposed (single colour background plane, graphics 1 and 2 plane). Video image sizes of WVGA or greater are supported by the digital interface signal outputs. Internally, the GLCDC acts as a GPX bus master function for accessing graphics data via the graphics GPX bus

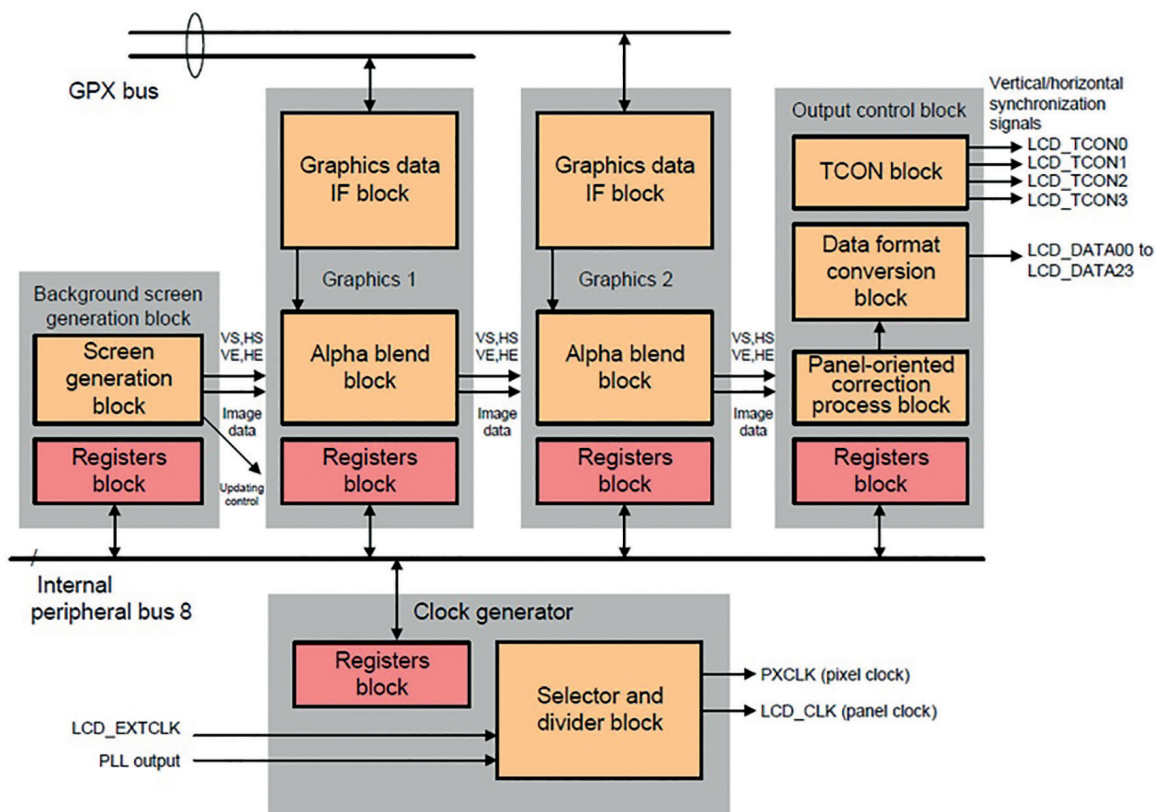


Fig. 28: GLCDC block diagram

Graphics data is provided by graphic modules like the 2D Drawing Engine (DRW) or the JPEG Codec. The DRW supports many object geometries of 2D objects such as lines, circles or triangles. Due to the integrated display list mode the CPU and the graphics controller can be decoupled efficiently and rendering can be performed in parallel with other CPU activities.

High-speed compression of image data and high-speed decoding of JPEG data can be done using the JPEG Codec. It complies with JPEG Baseline standard and is conform to ISO-IEC10918-2. Several pixel formats and images sizes can be configured very flexibly.

3. STARTER KITS

The microcontroller or any other smart component like an FPGA or a processor is the heart of the embedded system providing many features. To use it the controller has to be connected electrically and mechanically with the external world. Each MCU needs external components like capacitors or power supply for proper operation. Also the components to realize the desired functionality have to be added. In general the microcontroller and the additional components like resistors, ASICs or connectors are mounted on a PCB (Printed Circuit Board, Fig. 29 left).

The design and assembly of a PCB needs the knowledge of the system as well as of the components – and of course the expertise in circuit design including tools and methods. The basic steps for the circuit design including assembly of the components from requirement analysis to final assembly and testing are depicted in Fig. 29 and you need the expertise in all these steps to design a good and operational PCB.

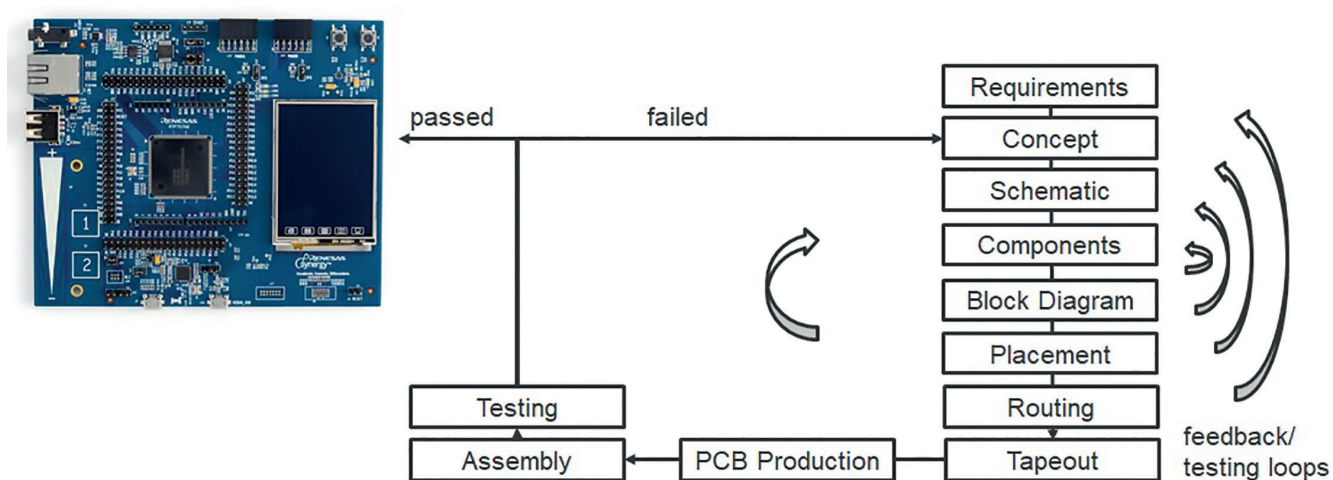


Fig. 29: PCB of the S7G2 Starter Kit (left) and basic design steps of PCB development

Besides the required expertise the hardware development is costly and needs a lot of resources – time, money, manpower. For the final hardware of a system this high effort is inevitable to get the best fitting solution. But maybe the application development should start rather early during the application development without a final concept, specification or hardware. Or you want to have a prototyping system. Or you just want to make some first steps with a new microcontroller or system. In such cases, the development of an own hardware and PCB will be much too costly. Instead the usage of a given hardware is preferred, even though it will not be the final and perfectly fitting solution. For this purpose so called Starter Kits, Evaluation Boards or Development Kits are provided by many (semiconductor) companies. These kits and boards are general purpose hardware generally dedicated to some main component, e.g. a microcontroller. It is used to provide as many features of the main component like the MCU as possible. Besides the main component these boards incorporate important external components and provide connections, connectors and interfaces to additional external components.

For microcontrollers the corresponding Starter Kits can be used to get started with the microcontroller – for example for students. Most pins of the microcontroller are available on the board (e.g. pin row connectors) for easy and flexible access. Some external components are included on the boards and already connected to the corresponding pins of the MCU. Connectors for power supply or interfaces are also part of the Starter Kit as well as standardized connectors for extension boards, e.g. PMOD™ connectors (Peripheral Module interface). Together with a development environment (refer to chapter 4) and sample programs it is a perfect and easy to use starting point to work with this MCU.

For more advanced users these Kits can be used for rapid prototyping of new systems and applications. The Starter Kit provides full flexibility due to the above mentioned family concept of microcontrollers. The software development can be done for the target microcontroller even without the final hardware and software and hardware development can be done in parallel. Also early software and application testing can be done. This parallelization of hard- and software development increases the reliability of the final application and product and speeds up the development significantly – shorter time to market!

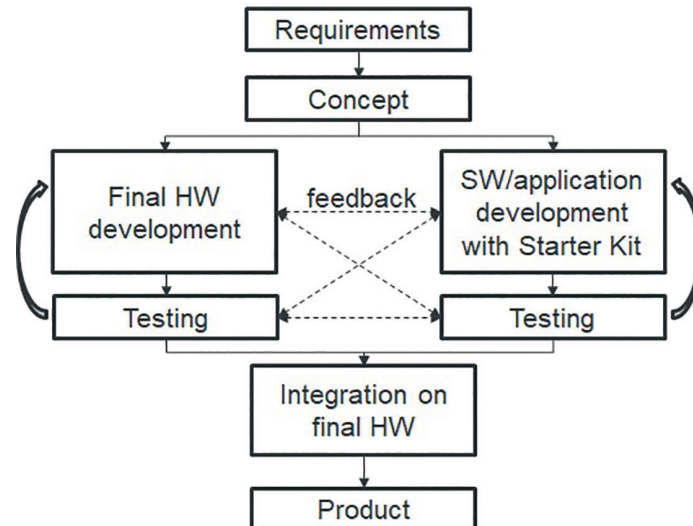


Fig. 30 : Parallel hard- and software development

3.1. S7G2 STARTER KIT

The S7G2 Starter Kit is a single board starter kit for Renesas Synergy™ S7G2 microcontroller (ARM® Cortex-M4 controller in 176 LQFP package) designed for first steps with Synergy, education and initial application development. Besides on-board components like a touch display or capacitive sensors it is expandable by external components using the pin row, PMOD™ connectors or an Arduino Shield compatible interface (Fig. 31). Configuration of the board and the connectivity is done by several jumpers. Most pins of the MCU are accessible on pin row connectors and besides the hardware you get the full Synergy support when using this Starter Kit: Synergy Software Package (refer to chapters 5 to 10), ISDE (chapter 4), software and application support and you can use all features of the Renesas Synergy™ platform (chapter 11). No additional hardware is needed to get started – just connect the Kit to your PC and start programming on API level using the ISDE.

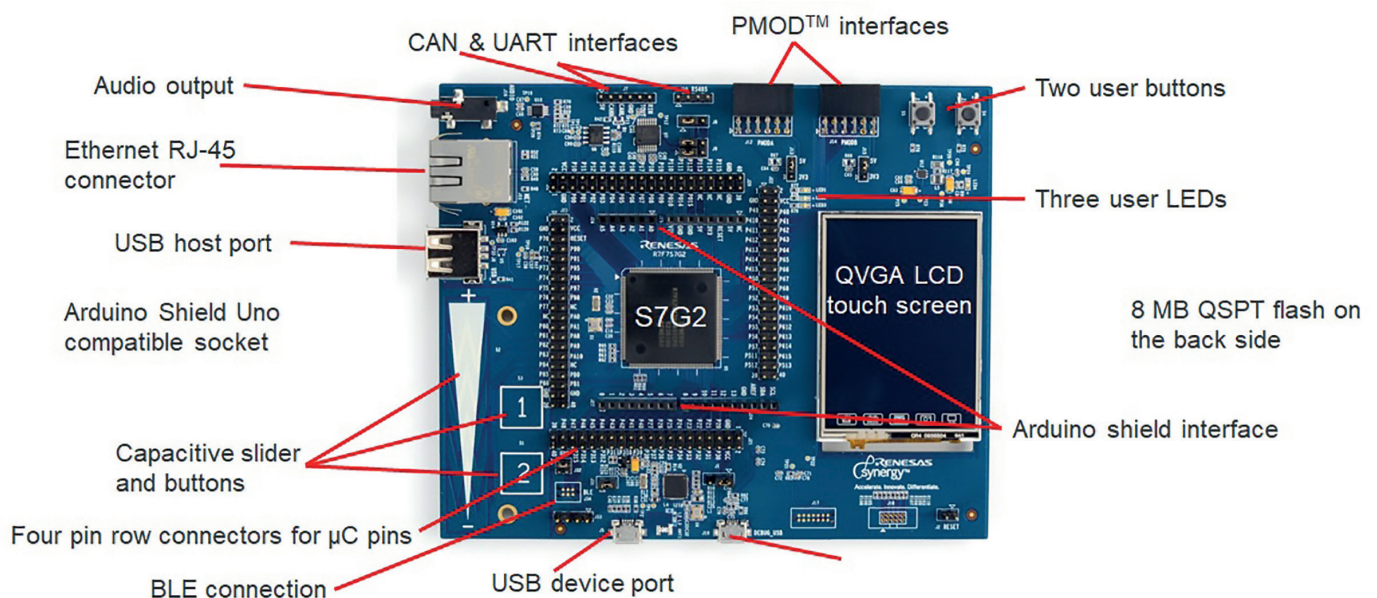


Fig. 31: Major components of the S7G2 Starter Kit

Timer modules consist of a set of timers (as the name implies...) and offer a high degree of flexibility to generate and measure signals and timings. In capture mode the timers capture input signals to measure interval times, periods or pulse widths. They can also be used to count external events or to generate events in compare mode. Pulses or pulse width modulation signals (PWM) can be generated in output mode, possibly using several timers.

Interface	
USB host port	High-speed (480 Mbps)
USB host port	Full-speed (12 Mbps)
Ethernet	10 Mbps and 100 Mbps)
LCD touch-screen	240 x 320 QVGA
PMOD™ connectors	PMOD™ A: SPI, three GPIO lines, interrupt line PMOD™ B: UART, three GPIO lines, interrupt line
Debug interfaces	JTAG emulation/debugging, SWGGER J-Link, JTAG/SWD
UART	RS-232 or RS-485 mode
CAN	

Table 4: Connectivity features of the S7G2 Starter Kit

To extend the features of the Starter Kit several extension interfaces are available. PMOD™ is an open standard interface for peripherals using 6 or 12 pin connectors in particular for prototyping and evaluation purpose. PMOD™ boards are small I/O interface boards that can be connected via the connectors quickly and easily without soldering. Many boards from many vendors with a PMOD™ connector are available, e.g. with an OLED display, a 3-axis MEMS accelerometer and gyroscope or a dual H-bridge motor driver.

Another simple way to expand the hardware of the S7G2 Starter Kit is by using Arduino Shields. Arduino is a very common HW/SW platform based on Atmel microcontrollers providing many solutions and functions. As it is easy to use and an open source project it is commonly used and a huge amount of functions and applications are available for everyone. The functionality of Arduino boards can be expanded by so-called Arduino Shields. Many of these expansion shields are available from many vendors covering functions like motor control shields, GPS shields or NFC/RFID shields. Due to a standardized Arduino Shield Interface these expansion boards can easily be plugged on top of the Arduino PCBs. Also the S7G2 provides an Arduino Shield compatible interface to mount extension boards on top of the Starter Kit without soldering.

The benefits of providing both PMOD™ connectors as well as an Arduino Shield compatible interface on the S7G2 board are manifold. They allow the simple, fast and cheap expansion of the Starter Kit and increase the flexibility of the board significantly due to the huge amount of available extension boards and functions.

In addition to the two standardized interfaces the Starter Kit also provides simple pin row connectors for flexible connection of external components and devices. Most pins of the MCU are available on the four pin row connectors. Using the pin row connectors enable the connection of devices and boards without a standard interface. For example in the lab (chapter 12) sensor boards with 3 pins (power, ground, analog signal) will be used and will be directly connected to the pin row connectors.

Besides the S7G2 board itself, the Starter Kit also contains an USB Type A to Micro-B cable and a quick start guide making the setup rather simple: just connect the board to your PC using the USB cable. This is an easy and fast way to access the entire Synergy Platform as you can use most of the SSP functions like the Integrated Solution Development Environment (ISDE, chapter 4), the real-time operating system (RTOS, chapter 7) or the application framework (chapter 8). For additional software, examples, further documentation or support, just check the single entry point for the whole Synergy world, the Renesas Synergy Gallery under <https://synergycastle.renesas.com> (chapter 11).

Using the Starter Kit with all the interfaces in conjunction with all the features of the SSP enables the rapid prototyping of own applications in a fast, reliable and cheap manner.

4. ISDE

After the hardware is now ready to use we need to program the microcontroller. The controller itself needs machine code to run – a list of basic instructions in binary code reflecting the required functionality. This functionality includes the basic configuration of the hardware and the modules, the interaction of the modules, the real-time requirements as well as the application. As binary code is hard to read, understand and program higher level programming languages like C or C++ are used. In addition, the development engineer wants to focus on his application, not on the low level programming. Therefore several tools are needed to link these two approaches (Fig. 32) and to translate the high level code to executable machine code. After the machine code is generated it has to be flashed into the memory of the microcontroller to run and debug the code.

The basic principles of software development is depicted schematically in Fig. 32. It contains several different tasks of programming and corresponding tools to finally get the machine code. For example the application engineer develops the application on a high level of abstraction. This development is done using dedicated tools like Matlab®/Simulink®. Afterwards the model is converted into C code by an automatic code generator. The C code file are translated into machine code using a suitable compiler and linker.

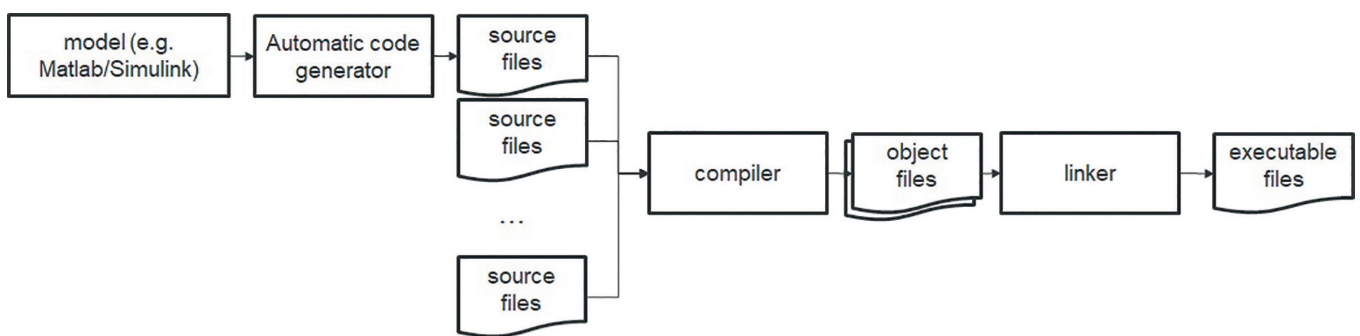


Fig. 32: Software development from high level design to machine code

The model part – here Matlab®/Simulink® and automatic code generation – depends mainly on the application and system under development and is independent from the following programming and hardware related parts (of course the application has to fit finally to the software and hardware...). Hence, the model part will always require a dedicated tool. But to simplify life for the programming part a single program covering the complete software development flow starting from C/C++ code is mandatory.

As depicted in Fig. 33 an IDE (Integrated Development Environment) includes all tools and programs for software development, from coding to debugging the running software. In general the IDE is a visual programming environment that uses standardized tools to focus on the application and software development and not on the tools. It also supports the developer with many extra features regarding usability, debug functionality and more.

The support for the developer starts at the very beginning – the coding using a text editor. Of course, any text editor can be used, but some extra features simplify the work and increase the reliability of the code, e.g. syntax colouring, highlighting and autocomplete for key words, code templates and more.

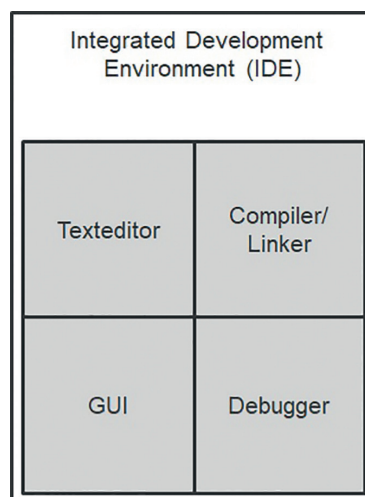


Fig. 33: Major components of an IDE

After the software is written in C/C++ it has to be translated into machine code and mapped to the hardware resources using a compiler and linker. Of course both the original C/C++ code and the machine code have to have the same functionality! During compilation and linking the code is analysed and the syntax of the code is checked. Feedback about errors (e.g. syntax errors) and warnings are reported. Also the code is optimized with regard to run-time, memory usage and dead code.

The binary code is then flashed to the microcontroller and the program can execute. Any code contains bugs (errors according to famous Grace Hopper), at least at the beginning (hopefully, it is more or less bug free in the final application...). A debugger is used for the analysis of the code and its behaviour and to find and eliminate the bugs to ensure correct functionality. For this purpose the debugger can control the program execution, e.g. by setting breakpoints or by single step execution of the code. During debugging data like register values, memory content or the machine code can be checked.

4.1. RENESAS e²studio

The development environment for the Synergy platform is Renesas proprietary ISDE (Integrated Solution Development Environment) e²studio. The term solution in ISDE emphasises the focus on the application due to many integrated tools and features within e²studio. It is based on Eclipse and Eclipse CDT.

Eclipse is a standard and common open-source development environment based on Java. A GUI displays several windows called views. Each view is associated with a task like navigation view or the editor. The different steps of the software development process are reflected in corresponding perspectives of the GUI. A perspective is an arrangement of tool bars, views and editors. Different perspectives with predefined and arranged views are available to fit to the different development steps like coding or debugging. Of course, the arrangement of views can also be changed in a very flexible manner. The development environment is extensible by additional plug-ins.

Eclipse CDT is specifically designed for C/C++ developers. It provides many features that support the C/C++ development like syntax highlighting, code generation, visual debugging tools, memory viewer, ...

The Synergy ISDE e²studio uses many features of Eclipse and Eclipse CDT and adds additional features like a smart manual. It is a GUI for code development, build (compiler and linker) and debug. It also provides numerous wizards for configuration and automatic code generation based on the SSP. Additional functions (or solutions, see ISDE) include integrated manuals providing device documentation and driver information and full support to enable application focused development.

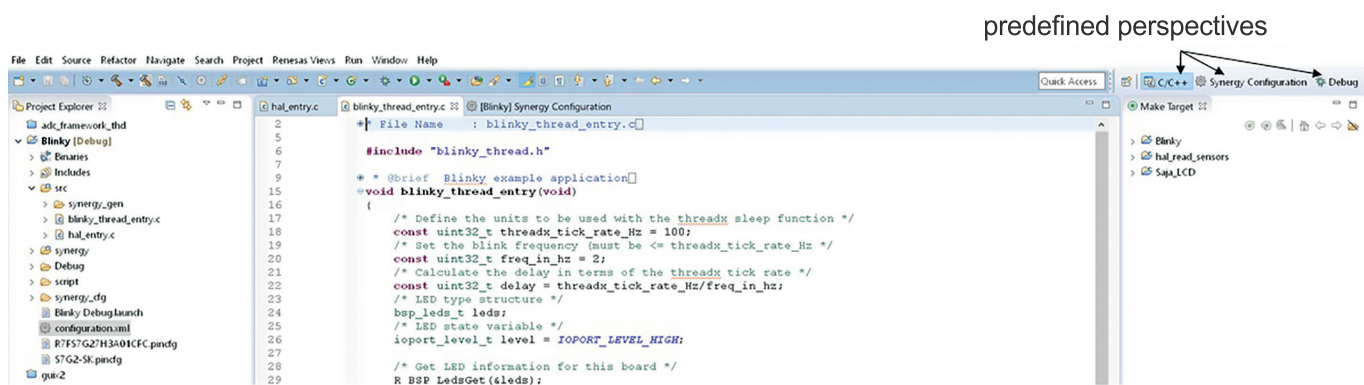


Fig. 34: Coding perspective of e²studio with folder structure, code and perspective selection

Some features of e²studio to simplify the developer's life and to enable fast and reliable code generation:

- Automatic code completion
- Keyword colour coding of source code
- Built in spell checker
- Powerful code navigation
- Comment and code folding options
- Jump to declarations
- C/C++ language style code formatting (auto indentation, brace matching, comment blocks etc.)
- Code templates
- Automated code constructs (if, while, do..while etc.)
- Auto completion for variable names, function names, struct/union members, define symbols, etc.

e²studio fits to the ARM[®] architecture of the Synergy MCUs and it uses the standard GNU Compiler Collection (GCC) as target build plug-in. Additional custom plug-ins are integrated for the Synergy Platform. The MCU and the SSP components can be configured graphically.

As the idea of the Renesas Synergy Platform is the single entry point and one shop solution, also third party components are integrated into e²studio and the SSP. For real-time application the RTOS (Real Time Operating System) ThreadX[®] by Express Logic is part of the SSP (chapter 7). The RTOS kernel-aware debugging is done within e²studio using TraceX[®]. For enhanced graphics the middleware GUIX[™] by Express Logic is integrated into e²studio (chapter 9). To design the graphics an external tool (GUIX Studio[™]) can be used which is available within the Renesas Synergy Platform. Fig. 35 depicts a schematic overview of the main components of e²studio. One major item (remember the more than 2000 pages of user's manual of the MCU) is the smart manual providing many information about functions, parameter, ... - on a simple click – without reading the user's manual...

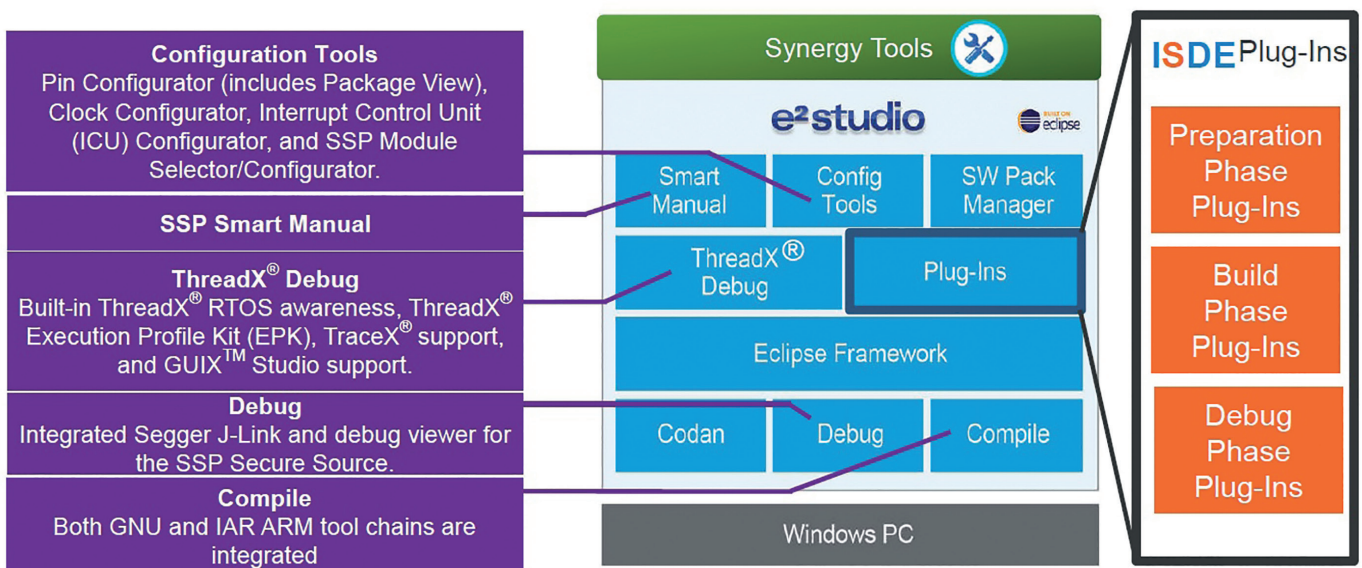


Fig. 35: Complete development within e²studio

Some major perspectives and features of the ISDE will be explained a bit more in detail in the following sections. During the lab (chapter 12) all these features and how to use them is done step by step.

In e²studio all applications are organized in Synergy projects. All projects within a workspace are displayed in the Windows-like Project Explorer view (Fig. 36 left). In general, one project is active at any time. For the different development steps e²studio provides several perspectives.

The basic perspective is the C/C++ perspective to enter and write your code. The editor in the middle of the C/C++ perspective provides a lot of support for coding. Features like code highlighting and autocomplete can be used and simplify the coding and readability of the code. A special feature is the smart manual (yellow box in Fig. 36). Just hover over parameters or instructions or functions and get detailed information and proposals. It is like a fully integrated user's manual and very intuitive.

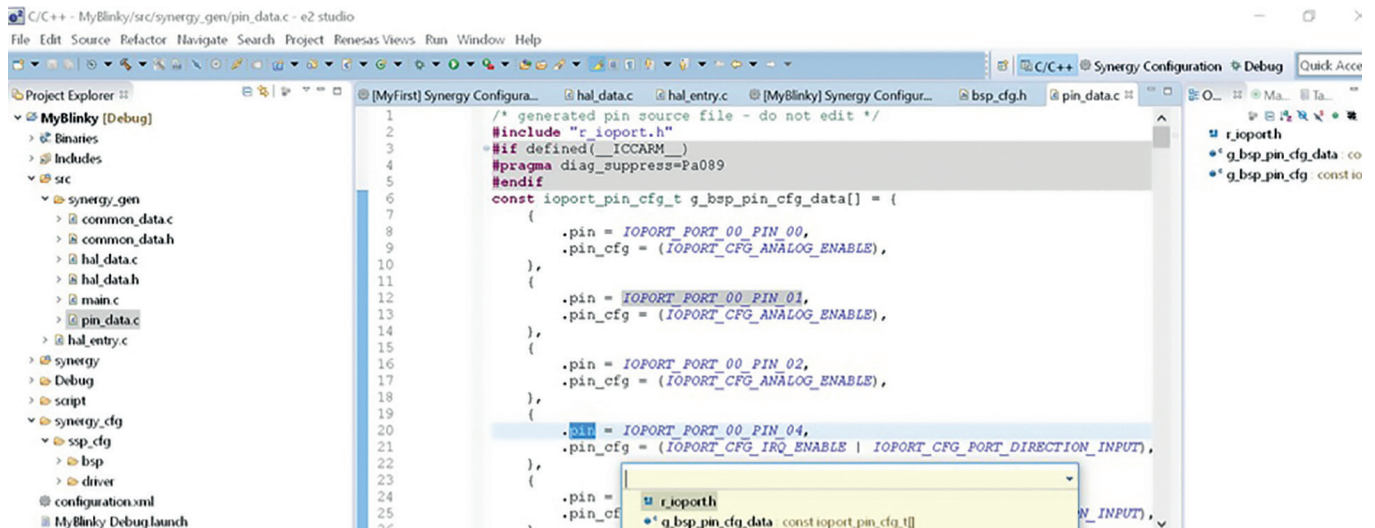


Fig. 36: C/C++ perspective of e2studio

Coding of the basic setup and configuration of the MCU and its peripherals and modules can be annoying and error-prone. A graphical approach is very intuitive for the developer. It is also easy to use without the need to read the hardware manual and hence supports a very reliable configuration of the MCU. In configuration perspective many basic settings can be done graphically. Fig. 37 depicts the pin configuration tab in C/C++ perspective for pin P000. Settings like mode, output type or pull-up resistor can be selected graphically. Besides the pin settings many other components can be configured in this way in dedicated tabs, like basic MCU settings (refer to chapter 5), clock tree, threads (refer to chapter 7) or messaging.

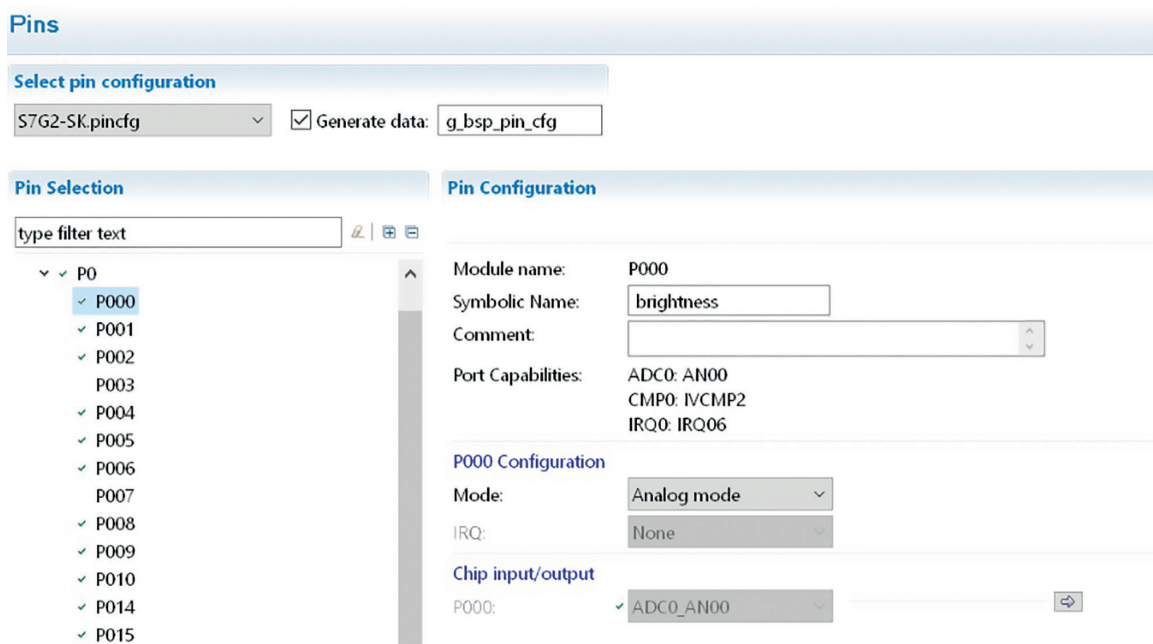


Fig. 37: Example for graphical configuration, here for pin P000

Another example for a basic configuration emphasises the benefits of a graphical solution. The clock tree of a MCU is a rather complicated topic. The MCU has many clock sources like external or internal oscillators. Using these clock sources several internal clocks have to be generated – for the CPU, the peripherals, the interfaces, the real-time clock, ... – depending on the application requirements. For the S7G2 27 registers for the Clock Generation Circuit to set up the clock tree according to the needs of the application. These 27 registers are described in detail on 34 pages of the user's manual. Besides just programming the registers, also restrictions, limitations and interdependencies of the clocks have to be considered. Hence, you need a detailed knowledge of the MCU and the hardware to set up the clock tree properly.

Therefore the configuration of the clock tree is difficult when done manually. Using the graphical configuration (Fig. 38) makes the settings rather simple. Just select the settings, all sources, generated clocks and the dependencies are displayed – that’s all for the configuration of the clock tree!

Clocks

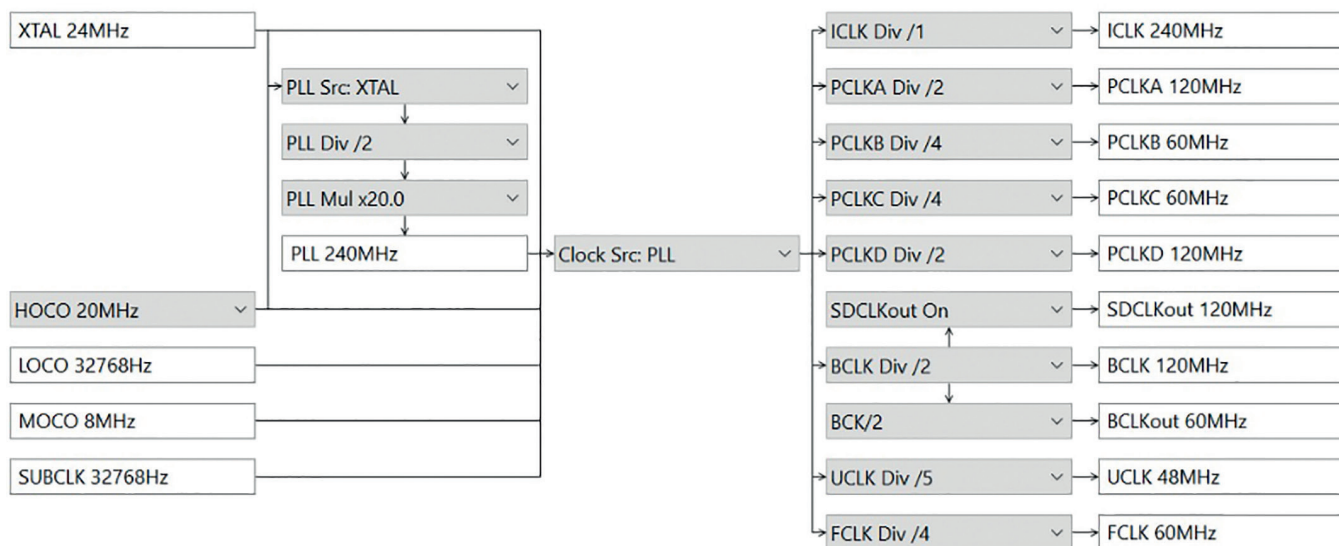


Fig. 38: Graphical configuration of the clock tree

Once the configuration is complete, C code can be generated – by simply clicking the “Generate Project Content” button in the C/C++ perspective. This automatically generated code is ready-to use and tested – great improvement of code reliability and speed up of development!

This automatically generated code configures the MCU to your needs – fast, simple, reliable. Now start to write your application code. For the clock tree the settings are stored in `synergy_cfg\ssp_cfg\bsp\bsp_clock_cfg.h`.

```
[guix2] Synergy Configuration  bsp_clock_cfg.h
1  /* generated configuration header file - do not edit */
2  #ifndef BSP_CLOCK_CFG_H
3  #define BSP_CLOCK_CFG_H
4  #define BSP_CFG_XTAL_HZ (24000000) /* XTAL 24000000Hz */
5  #define BSP_CFG_PLL_SOURCE (CGC_CLOCK_MAIN_OSC) /* PLL Src: XTAL */
6  #define BSP_CFG_HOCO_FREQUENCY (2) /* HOCO 20MHz */
7  #define BSP_CFG_PLL_DIV (CGC_PLL_DIV_2) /* PLL Div /2 */
8  #define BSP_CFG_PLL_MUL (20.0) /* PLL Mul x20.0 */
9  #define BSP_CFG_CLOCK_SOURCE (CGC_CLOCK_PLL) /* Clock Src: PLL */
10 #define BSP_CFG_ICLK_DIV (CGC_SYS_CLOCK_DIV_1) /* ICLK Div /1 */
11 #define BSP_CFG_PCKA_DIV (CGC_SYS_CLOCK_DIV_2) /* PCLKA Div /2 */
12 #define BSP_CFG_PCKB_DIV (CGC_SYS_CLOCK_DIV_4) /* PCLKB Div /4 */
13 #define BSP_CFG_PCKC_DIV (CGC_SYS_CLOCK_DIV_4) /* PCLKC Div /4 */
14 #define BSP_CFG_PCKD_DIV (CGC_SYS_CLOCK_DIV_2) /* PCLKD Div /2 */
15 #define BSP_CFG_SDCLK_OUTPUT (1) /* SDCLKout On */
16 #define BSP_CFG_BCK_DIV (CGC_SYS_CLOCK_DIV_2) /* BCLK Div /2 */
17 #define BSP_CFG_BCLK_OUTPUT (2) /* BCK/2 */
18 #define BSP_CFG_UCK_DIV (CGC_USB_CLOCK_DIV_5) /* UCLK Div /5 */
19 #define BSP_CFG_FCK_DIV (CGC_SYS_CLOCK_DIV_4) /* FCLK Div /4 */
20 #endif /* BSP_CLOCK_CFG_H */
21
```

Fig. 39: Clock settings in `bsp_clock_cfg.h`

After all the software is ready – configuration as well as application software – it should of course run on the MCU. So, in the next step we have to compile, build and flash the code to the memory of the microcontroller. To provide as much insight into the code and the MCU during debugging the debug perspective provides a dedicated set of windows, e.g. to check the code, set breakpoints or view variables and registers. Run your code and see what happens...

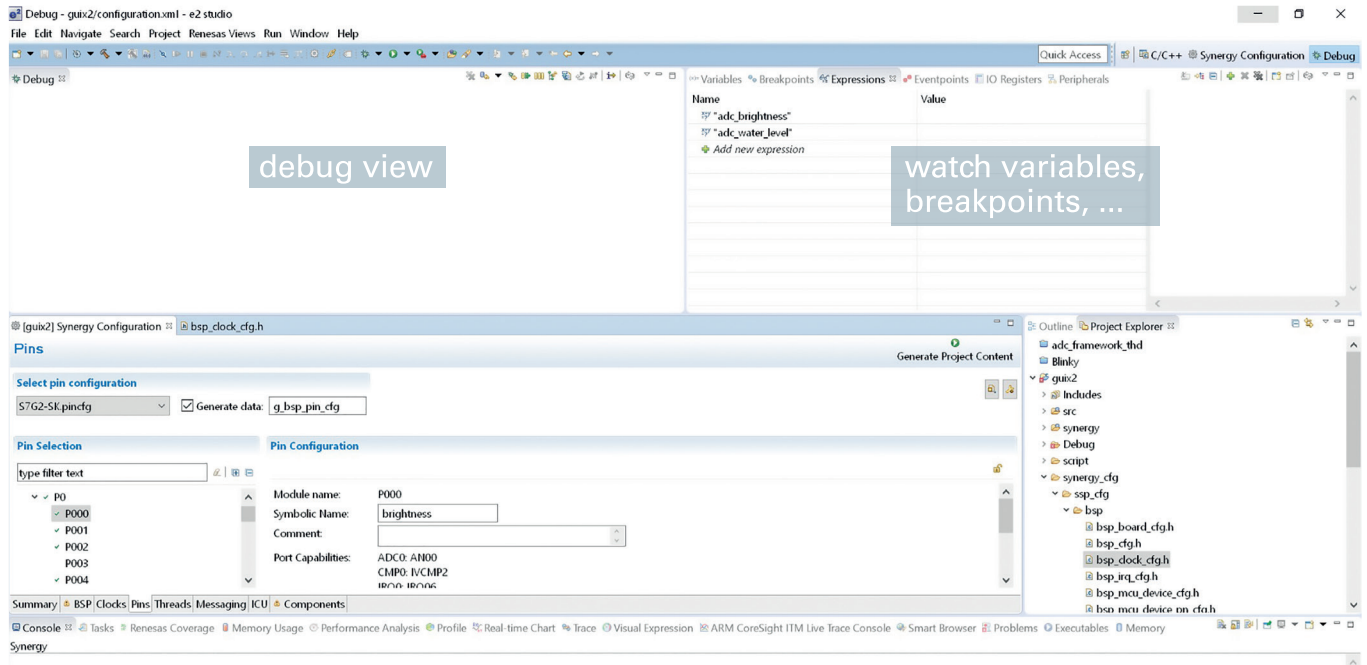


Fig. 40: Debug perspective

In the final application the microcontroller is the key element that has to fit to the application. But for the selection of the MCU it is not just the hardware that decides, but also everything around the hardware. In particular, the available tool set and support is in the end a key differentiator, so the ecosystem is very important. Hence consider the complete MCU setup for your next project – hardware, available tools like the IDE, software and support.

5. BOARD SUPPORT PACKAGE

In the previous chapter the support for the developer provided by the ISDE was introduced, like graphical configuration, automatic code generation and much more. Programming of the basic configuration and the setup of the MCU is the basis for all applications. Before running any user specific application code (famous `main()` function in C) the setup of the MCU has to be done. Besides configuration of clocks, ports and other modules also the C runtime environment has to be set up including stack setup, heaps or the initialization of the RAM. This is an interesting job for an embedded systems engineer or hardware engineer, but maybe not for an application engineer.

Even with the features of the ISDE the basic configuration of the MCU takes some time, needs some effort and is error-prone to some extent. Just consider the configuration of all the MCU pins: the MCU is mounted on a PCB and the pins are connected to the outer world. Some are used as input, some as output, digital or analog. Some are connected to external ICs or other components and have to provide the corresponding function of the internal peripheral modules. For the Synergy S7 MCU of the S7G2 Starter Kit there are 176 pins to be configured. Even with the graphical approach this work is annoying and time consuming – and you would like to focus on your application development, not on basic MCU configuration.

So, it would be great to get even more support to bring the MCU as fast and simple as possible from reset to the application code. For this extended support the knowledge of the MCU and the external hardware is needed. The semiconductor company of the MCU knows best about the MCU and the board developer knows best about the hardware. Hence the expertise of the suppliers of the MCU and the PCB can be used to get the initial configuration code. Together with subsequent automatic code generation shortens the configuration time significantly and increases reliability.

Automatic code generation for the initialization software is very common for MCUs, provided by most semiconductor companies, e.g. Renesas' Applilet for Renesas proprietary MCU cores like RX and RL78, Infineon's Dave or STM's STM32CubeMX. The companies use their expertise to support the customers with smart solutions for the low level drivers. Besides the basic initialization software these tools also configure peripheral modules graphically. A simple GUI can be used for the configuration and you get the reliable code without writing a single line of code yourself – even without the detailed knowledge of the underlying hardware.

So far, we have the chance for graphical configuration of the MCU. But still the configuration has to be done by the developer and in addition, there is the surrounding hardware that has an impact on the MCU settings. So still some work to do. To include also the hardware and board specific settings a BSP (Board Support Package) can be used. A BSP incorporates all the settings and configurations for all modules and peripherals taking the hardware into account. Each BSP is customized to a MCU and a board and makes it possible to generate all start-up code automatically. Of course again without writing a single line of code, but this time even without clicking the configuration and knowing the hardware. Just select the corresponding BSP, generate the code and start focussing on your application as the BSP configures the MCU and provides low level drivers.

5.1. SSP BSP

The Renesas Synergy BSP configures the MCU according to the requirements of the corresponding board, e.g. the S7G2 MCU for the S7G2 Starter Kit. In general each board provided by Renesas is supported by a fitting BSP. The BSP generates the code to get the MCU from reset to the user code/`main()`. It is directly integrated into the ISDE e²studio which makes the usage as simple as possible. As the core of the BSP is CMSIS (ARM® Cortex® Microcontroller Software Interface Standard) compliant it follows the requirements of the standard and the naming conventions. The BSP is the basis for all Synergy projects, but of course the used can change all setting of the BSP afterwards using the graphical configuration tabs. Full flexibility combined with simple usage.

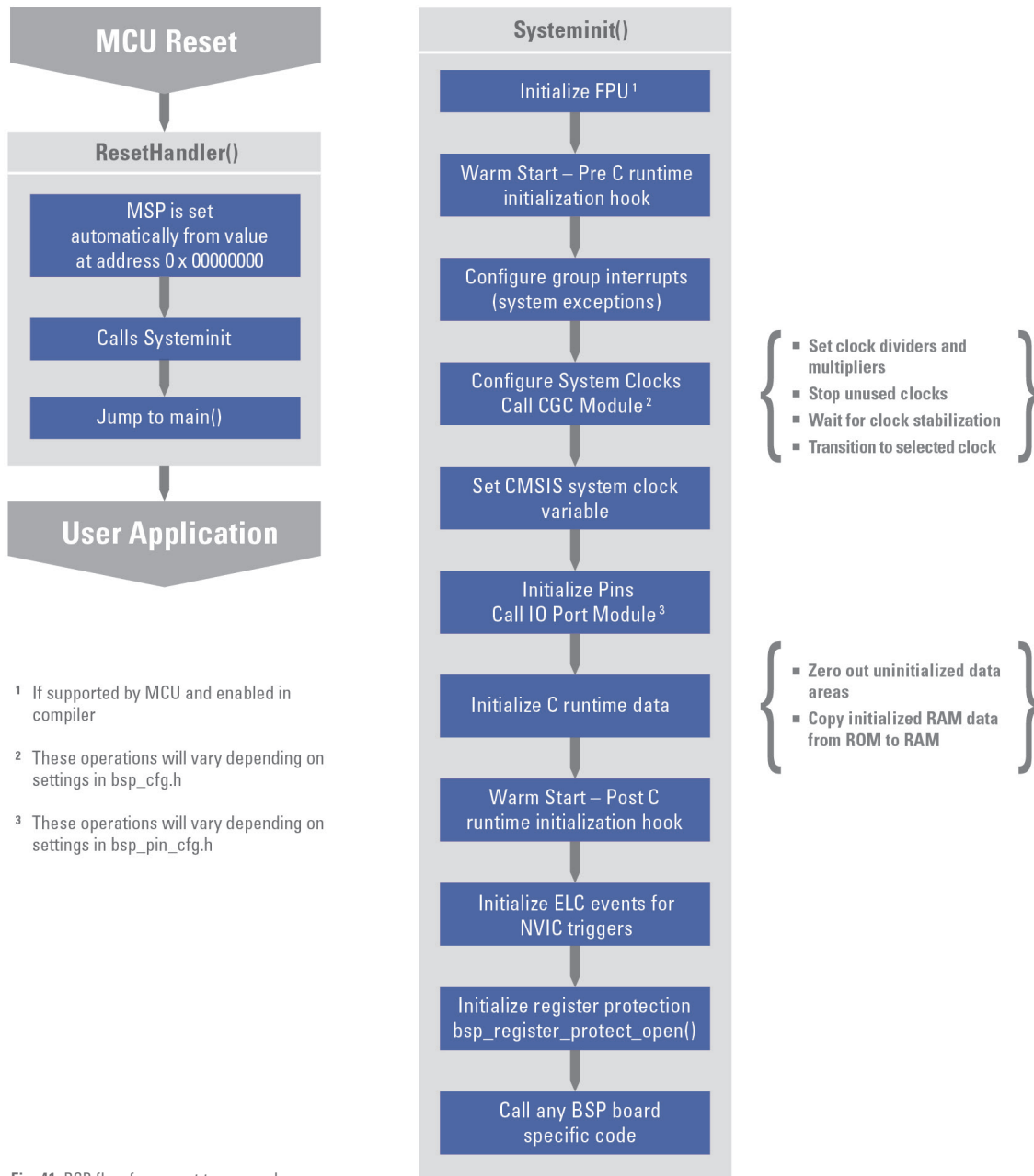


Fig. 41: BSP flow from reset to user code

Fig. 41 shows the flow and actions of the BSP from reset to main(). It sets up the stacks, clocks, interrupts and the C runtime environment. Within the PreC runtime initialization hook clocks are set and functions like safety code, memory check are performed and the user can request a callback. The PostC runtime initialization hook performs functions like full speed tests or ADC diagnostic.

The BSP is heavily data driven and configuration files (remember the Window-like project explorer view of the ISDE) contain the configuration code. The configuration files are located in the workspace directory (...workspace\MyBlinky\synergy_cfg\ssp_cfg\bsp) and are fully visible. But don't change these files manually as they will be updated according to the settings within the ISDE with every click on the Generate Project Content button. This code is as usual generated by the ISDE when Generate Project Content button is clicked. The configurable BSP settings include RAM, ROM and flash sizes, main stack and heap size, RTOS enable and more.

The BSP is the bottom layer of the Synergy Software Package. It provides public functions like interrupt handling, register protection or clearing flags with a strict and standardized naming convention. All functions start with R_BSP and macros start with BSP_. These public functions build the API (Application Programming Interface) of the BSP. Application code (or higher software layers) can use these API for direct access to the BSP features. The R_CGC_ClocksCfg function is depicted in Fig. 42 as an example. It configures the clock tree and is called during the start-up to set all clocks. Application code can call this function at runtime to change the clock settings – if needed in the application.

In general an API is a set of functions of a module to provide services to higher level software. It can be called by the higher level software to use the functionality of the modules. Using the API separates the hardware programming from the application as it is an abstraction of the hardware. The complexity of the hardware is encapsulated by the API keeping full control of functionality by the application and user. Don't care about the registers and hardware any more, just use the functions provided by the API. Details of all functions provided by the API are available within the smart manual of e²studio making the usage of the API as simple as possible and simplifies the life for the developer – keep your focus on application! Other examples for BSP API function include the initialization of a defined lock structure (R_BSP_SoftwareLockInit()), the stop of specified modules (R_BSP_ModuleStop) or the enabling of register protection (R_BSP_RegisterProtectEnable).

```

198
200
213 * * @brief Reconfigure all main system clocks.
214 *ssp_err_t R_CGC_ClocksCfg(cgc_clocks_cfg_t const * const p_clock_cfg)
215 {
216     ssp_err_t err = SSP_SUCCESS;
217     cgc_clock_t requested_system_clock = p_clock_cfg->system_clock;
218     cgc_clock_cfg_t * p_pll_cfg = (cgc_clock_cfg_t *) &(p_clock_cfg->pll_cfg);
219     cgc_system_clock_cfg_t sys_cfg = {
220         .pelka_div = CGC_SYS_CLOCK_DIV_1,
221         .pelkb_div = CGC_SYS_CLOCK_DIV_1,
222         .pelkc_div = CGC_SYS_CLOCK_DIV_1,
223         .pelkd_div = CGC_SYS_CLOCK_DIV_1,
224         .bclk_div = CGC_SYS_CLOCK_DIV_1,
225         .fclk_div = CGC_SYS_CLOCK_DIV_1,
226         .iclk_div = CGC_SYS_CLOCK_DIV_1,
227     };
228     cgc_clock_t current_system_clock = CGC_CLOCK_HOCO;
229     g_cgc_on_cgc.systemClockGet(&current_system_clock, &sys_cfg);
230
231     cgc_clock_change_t options[CGC_CLOCK_NUM_CLOCKS];
232     options[CGC_CLOCK_HOCO] = p_clock_cfg->hoco_state;
233     options[CGC_CLOCK_LOCO] = p_clock_cfg->loco_state;
234     options[CGC_CLOCK_MOCO] = p_clock_cfg->moco_state;
235     options[CGC_CLOCK_MAIN_OSC] = p_clock_cfg->mainosc_state;
236     options[CGC_CLOCK_SUBCLOCK] = p_clock_cfg->subosc_state;
237     options[CGC_CLOCK_PLL] = p_clock_cfg->pll_state;
238
239     #if CGC_CFG_PARAM_CHECKING_ENABLE
240     CGC_ERROR_RETURN(HW_CGC_ClockSourceValidCheck(requested_system_clock), SSP_ERR_INVALID);
241     CGC_ERROR_RETURN(CGC_CLOCK_CHANGE_STOP != options[p_clock_cfg->system_clock], SSP_ERR_INVALID);
242     #endif
243     if (CGC_CLOCK_CHANGE_START == options[CGC_CLOCK_PLL])
244     {
245         CGC_ERROR_RETURN(HW_CGC_ClockSourceValidCheck(p_pll_cfg->source_clock), SSP_ERR_INVALID);

```

Fig. 42: R_CGC_ClocksCfg function for configuration of all system clocks

During setup of a new project in e²studio one of the first items to select is the BSP you want to use for your project (refer to the lab chapter 12). For all available boards (at least all boards supported by Renesas) the corresponding BSP can be selected during project configuration. The selection of the BSP of course also selects the correct MCU of the board. For example the S7G2 Starter Kit has its own BSP and uses the R7FS7G27H3A01CFC – the MCU mounted on the Starter Kit.

After the project is set up, the BSP configuration tab can be found in the configuration perspective of e²studio – just next to the pin, clocks or threads configuration tab. As usual in configuration perspective, the properties tab in the lower left corner lists all the setting for the selected item, here for the BSP and the MCU. Some settings for the BSP and MCU displayed in the properties tab include memory sizes (flash, ROM, RAM), package type and pin number, voltage, watchdog timer settings and many more.

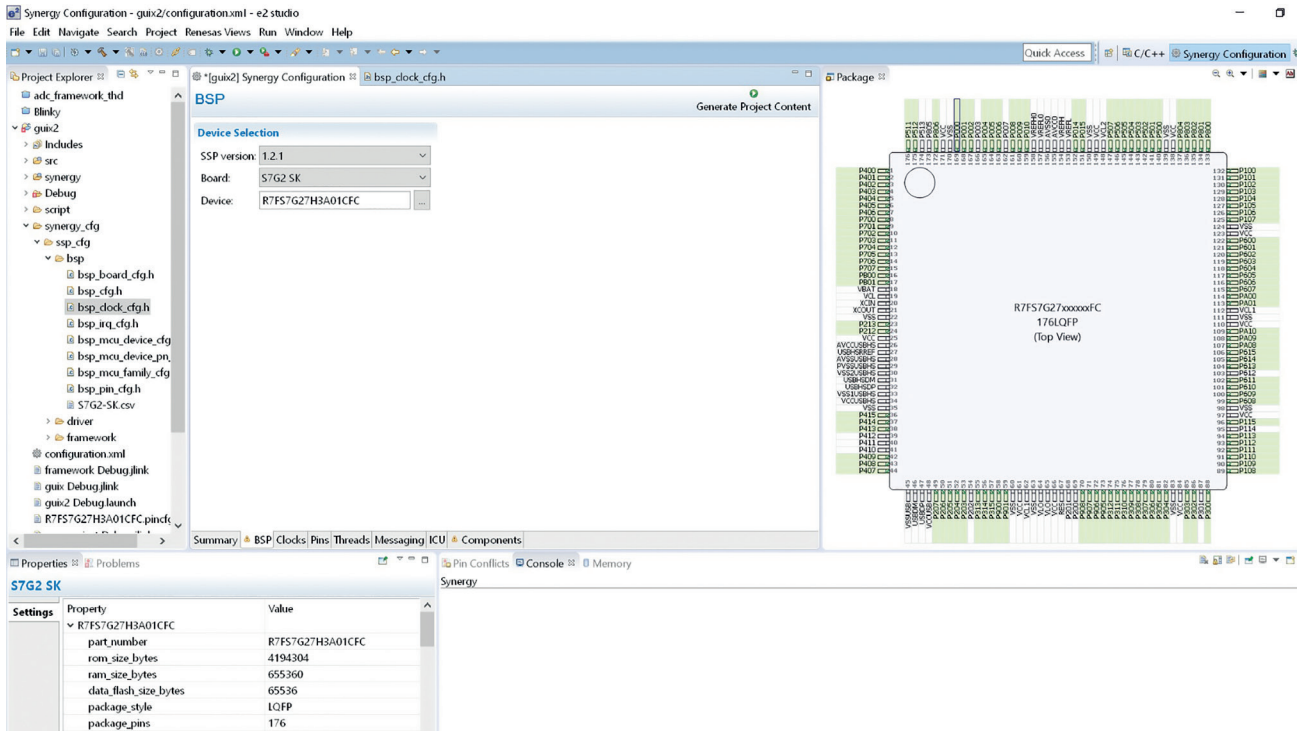


Fig. 43: BSP configuration tab

Even if the configuration is done graphically and is thus rather simple and intuitive there is of course still the possibility to make mistakes. Therefore the ISDE also provides consistency checks wherever possible. E.g. for the pin configuration a conflict checker runs to avoid errors like multiple allocation of a pin by different peripheral modules. Any inconsistency is displayed and commented to find the problem. This consistency checking increases the reliability of the settings and hence of the initialization code.

On the right side of the configuration perspective the pinout of the selected MCU is shown with all the configured pins. The consistency check is done automatically to show any problems and inconsistencies of the pin settings. If everything is green – go ahead.

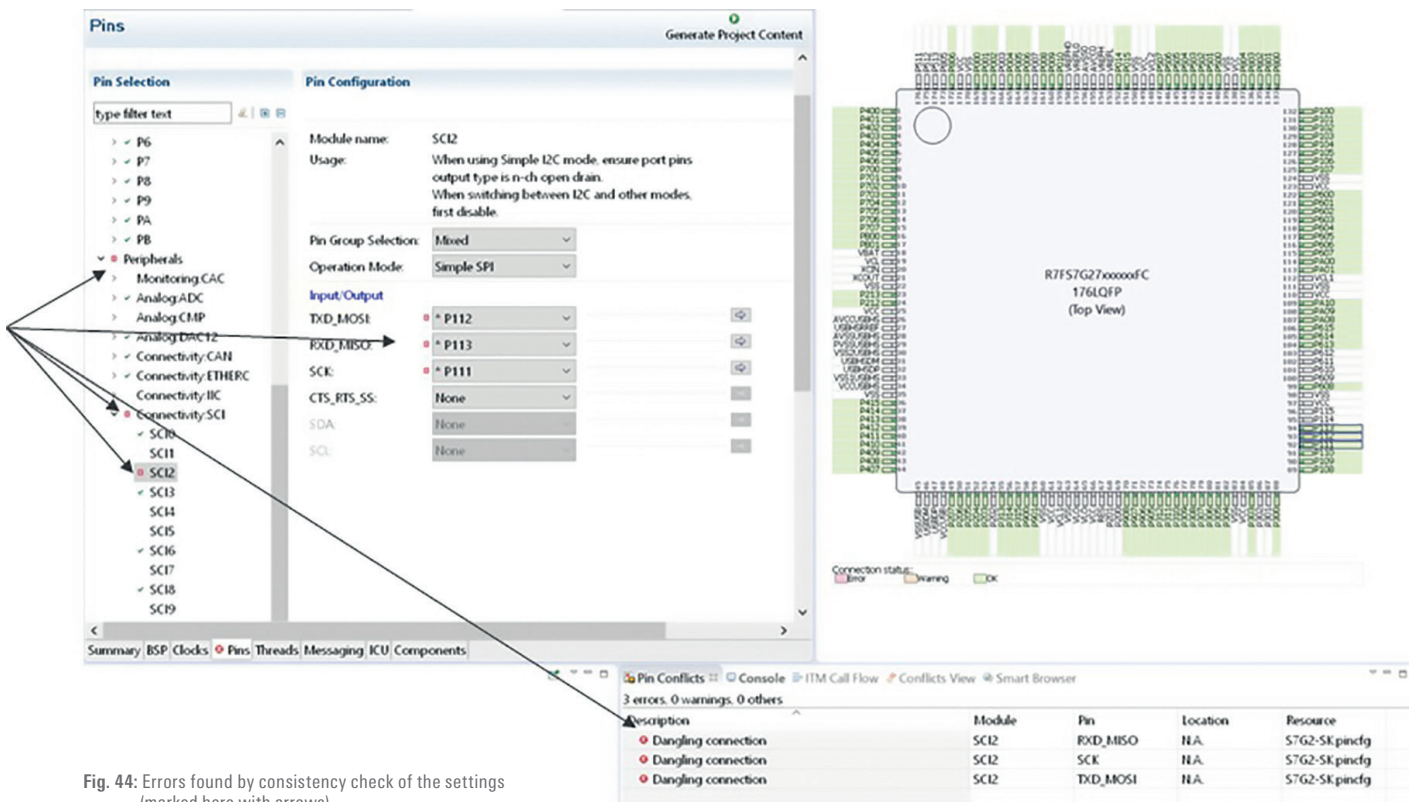


Fig. 44: Errors found by consistency check of the settings (marked here with arrows)

The concept of the BSP provides many benefits to support the developer like automatic settings, a low level API and the generation of start-up code. This start-up code from BSP of course requires some memory space of the MCU, both RAM and flash memory. As memory is one of the major limitations of MCUs and for embedded systems this code size of the start-up code should be as small as possible. The exact size depends on the used BSP and the used compiler and options. For some BSP the required flash size is listed in Fig. 45 for the GCC compiler (that is also used during the lab). For the S7G2 BSP the start-up code is less than 6 kB.

Table 14.1 Memory Usage for Board Support Package – GCC Compiler

able 1	BSP	Flash (Bytes)
	DK-S7G2	5,577
	DK-S3A7	5,093
	DK-S124	3,245
	PK-S5D9	4,888

Fig. 45: Memory usage of BSP

As mentioned the BSPs are available for existing boards. What happens if you want to have your own board? Just generate your own BSP within e²studio – as you know best about your board! It's just a few steps to configure the MCU according to your board and application and automatically generate the start-up code afterwards:

- Create a new Synergy C Project in e²studio
- Select Customer User Board (Any Device) and corresponding MCU
- Customize all tabs (BSP, Clocks, ...) to match your board
- Generate Project Content and build project
- Export custom board pack by right-clicking on the project and selecting Export Synergy User Pack
- Create a board pack and follow the export steps

This time of course you have to know or figure out the correct MCU settings by yourself. But after you have done it on your own, BSP is ready to use.

6. HARDWARE ABSTRACTION LAYER

The MCU on the board is now configured correctly by the BSP and the programming of the user code can start. But what is your user code? On the one hand you have your application, e.g. some control algorithm developed in Matlab®/Simulink®. These algorithms in general don't care about details of the underlying hardware. On the other hand you have to program and use the peripheral modules of the MCU to realize the required functionality. So in the end writing the user code will be a mixture of application related code and hardware related code. And this confusing mixture is a mess for your user code. In particular for complex systems. Therefore let's have some thoughts about software and coding (at least to some basic extend...) and find a more clever way of coding.

What is essential for good code? Okay, the answer to this question will depend strongly on whom you will ask... But the first requirement of course is that the code works – rather obvious. A reliable code that always works is the basis for all the other items.

To prove that it works it has to be tested and testability of the code is strongly required. Keep in mind: everything that is not tested will not work – or will at least have a high risk that it does not work. So, try to set up a smart test environment, test plan and test suite for proper testing. To find and eliminate bugs the code should be easy to debug. Otherwise, error finding will be like looking for a needle in a haystack.

To increase efficiency of software development it makes sense to reuse proven and reliable code – don't reinvent the wheel every time. To enable reusability the code should also be portable to run on different hardware – e.g. on different Renesas Synergy MCUs. Hence, a simple change or exchange of the hardware related code is highly appreciated. Depending on the application the lifetime of a system can be rather long and during the lifetime updates of the software might be necessary. This long lifespan of the software together with the updates requires a software that is easy to maintain – not only by the developer of the software but also by other software engineers.

Besides flexibility with regard to the underlying hardware, the code should also be flexible regarding the adaption to changing needs and requirement of the application and system. This flexibility can be supported by a code that is as simple as possible and user-friendly, e.g. clearly structured and high degree of readability. Last but not least it should be possible to open the code for distributed collaboration.

Taking all these items and requirements into account it is clear that a mixture of hardware related code and application code is possible but not a good idea. So let's recap the approach of the BSP: the basic interaction with the hardware like programming on register level is done by predefined functions of the BSP. These functions build the API of the BSP and provide some degree of abstraction of the complexity of the hardware. Software on top of the BSP can call the predefined functions instead of register programming. A similar approach with a higher degree of abstraction is done with the HAL (Hardware Abstraction Layer) as the name already implies.

Basis for the HAL is a component-based software architecture (Fig. 46). In this architecture, the software is split into individual functional or logical components or modules. Each module encapsulates a set of related and dedicated functions. This split results in a modular software with a high level of abstraction. The modules communicate via interfaces and provide their functions to other modules – the API of the module. Also the module may require itself some functions of a lower level module.

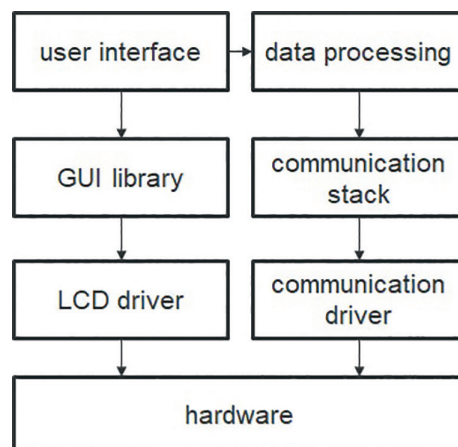


Fig. 46: Schematic of component-based architecture, arrows represent interfaces

Modular software is very popular as the modularity provides many benefits. As the modules have a well-defined functionality and interfaces they can be reused in different applications or systems without any effort. For the same reason the replacement of modules with similar functionality is rather simple. As the encapsulated functionality is just accessible by the interface, there is no need to exactly know the internal processes of the module.

As already mentioned, each module has a standardized interface and requires services from and provides services to other modules. The provided functions of the module are accessed by function calls. This API again abstracts the underlying implementation and provides simple to use functions for higher level software modules.



Fig. 47: Basic view of a module with interfaces (left) and application code using a module

In addition, the modules are independent from each other and can be combined to generate more complex components. Fig. 48 depicts an example from the SSP for a stack of modules to build a more complex application. One part of the application realizes an audio playback function and uses the audio playback module and the corresponding API. The audio playback module itself requires a data transfer module – but the application does not care about which data transfer module is used. In this case, the audio playback module uses the functions of the DTC module (Data Transfer Controller). Any other suitable data transfer module is also possible, just change the transfer module. The other parts of the application use an UART driver and a SD Card Driver. These two modules also need a data transfer module – just use the same DTC module to keep the code small, smart and reusable.

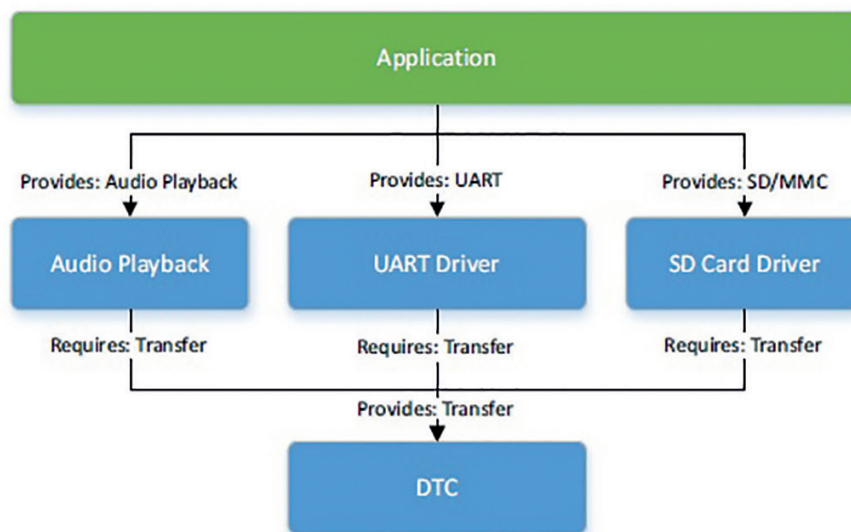


Fig. 48: Stack of modules for an audio playback function

The benefits of the component-based architecture using modules with defined functions and interfaces are manifold:

- Exchangeability
- Reduced cost by reuse
- Simpler development
- Reliability
- Maintenance
- Evolution of implementation
- Extendable
- Flexibility

The use of modules makes the life for the application developer much easier. A simplified schematic is shown in Fig. 49 taking also an operating system (refer to chapter 7) into account. The application code uses some components and these components interact themselves with the BSP or the operating system. From the usability point of view, this approach is already very good, as the application does not interact with the BSP or the hardware directly. But the portability is rather poor: if the hardware or the operating system changes, also the components have to be changed as they include BSP and hardware related parts.

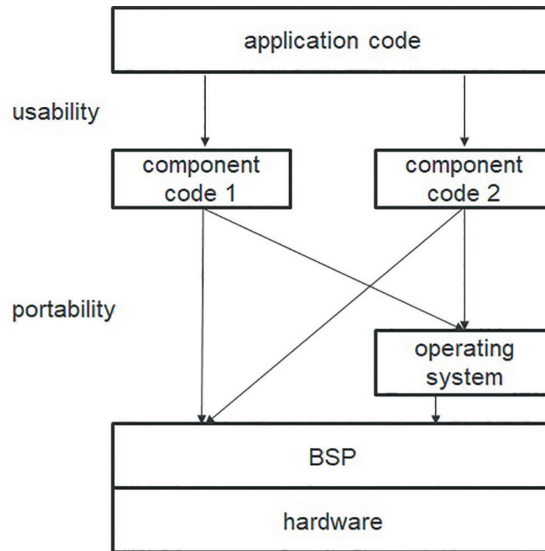


Fig. 49: Components acting directly with BSP and operating system

To increase the portability of the code one idea is to add an intermediate module to each component that can directly interact with the BSP and hardware (left of Fig. 50). If the hardware changes just change the interface module and leave the component as it is. Portability to different hardware is better with this approach as the component code does not contain BSP and hardware parts but many interfaces to the hardware are needed.

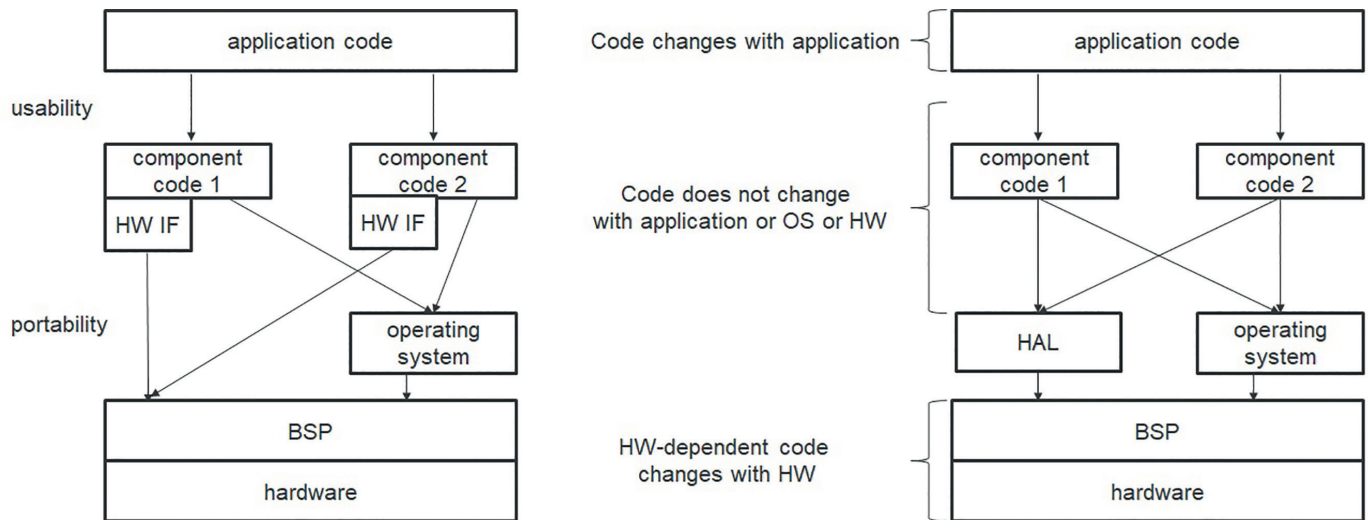


Fig. 50: Components with hardware interface for better portability (left); HAL layer architecture (right)

To reduce the amount of interface modules to just one the Hardware Abstraction Layer (HAL) is used. This extra layer separates the component code completely from the hardware and BSP related part and provides maximum portability. If the hardware or BSP changes, the component code remains untouched. The HAL hides the complexity of the hardware for the application components and enables a high level of abstraction and a high degree of flexibility. The application code depends of course on the application – not on the hardware. The component code is independent from application or operating system or the hardware.

There are many advantages of using an HAL: the flexibility to change the hardware is very high as both the application and the component code is independent of the hardware. It makes reuse of software much simpler due to the modular setup and hence speeds up the development, reduces coding time, increases

the efficiency and reduces the hardware dependence. Due to the reuse of existing components the reliability increases and the code will have less bugs – both during testing reducing the effort as well as in the final code.

To make use of the HAL for the application development in an efficient and reliable way the HAL has to fulfil some requirements:

- Dedicated to hardware
- Function has to be proven
- Standardized APIs
- Bug free
- Excellent documentation

If these requirements are met by the HAL it increases the abstraction for the application development, increases reliability and efficiency and puts the focus on the application, not on the hardware coding.

6.1. SSP HAL

The Synergy HAL is custom-built for the Synergy MCUs and fulfils the requirements for HAL. It has a proven and guaranteed functionality and contains no bugs – okay, no complex technical system can be proven to be completely bug free, but any bug is immediately fixed by Renesas. All functions and interfaces are well documented in the Renesas Synergy™ Software Package User's Manual. Just have a look, just about 2500 pages... But no worry, the ISDE provides some nice features to simplify the development, like autocomplete and smart manual.

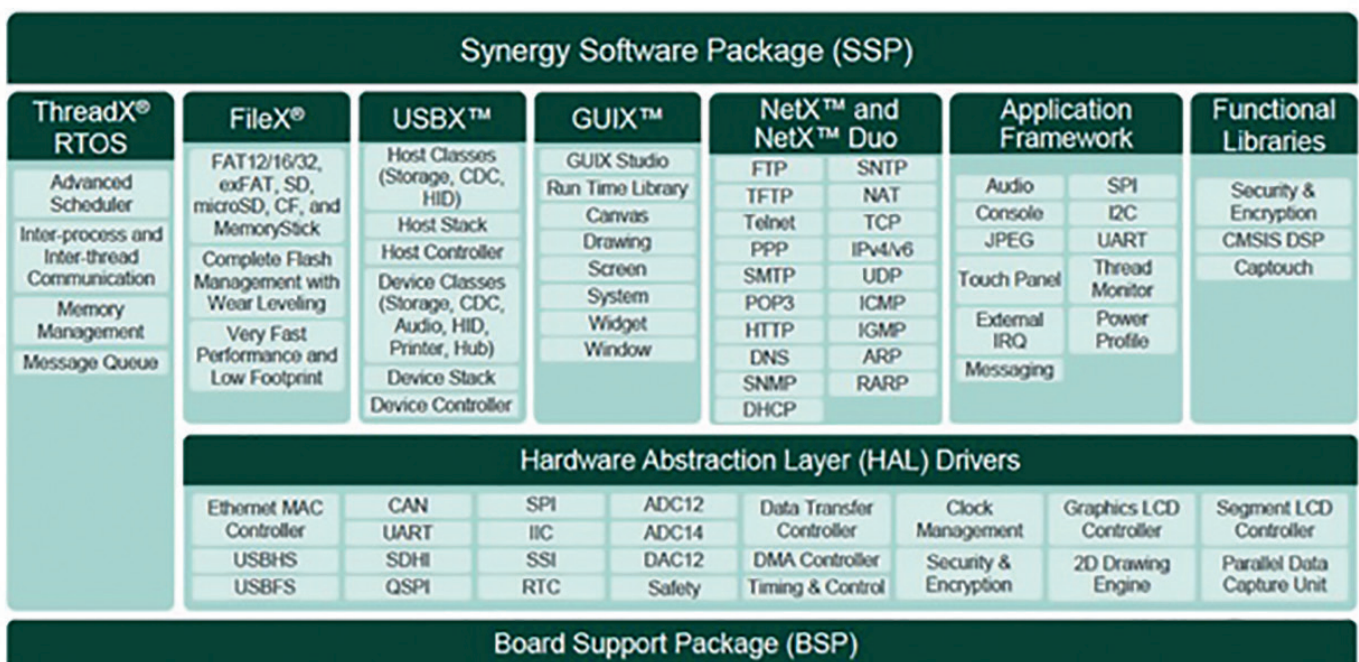


Fig. 51: Synergy SSP overview with HAL

The HAL drivers are device independent and well defined drivers for peripherals of Synergy MCUs. The HAL is located on top of the BSP and it uses the API of the BSP to interact with the hardware. For higher layer components like middleware, functional libraries or the application code the HAL provides an API itself. If really required the higher layer components can also bypass the HAL and access the BSP or hardware directly.

The underlying functionality of peripheral interfaces can be implemented by multiple devices drivers. This allows an extensible configuration of underlying hardware:

- Some peripherals support multiple interfaces
- Some interfaces are supported by multiple peripherals
- Some peripherals have a one-to-one mapping to an interface

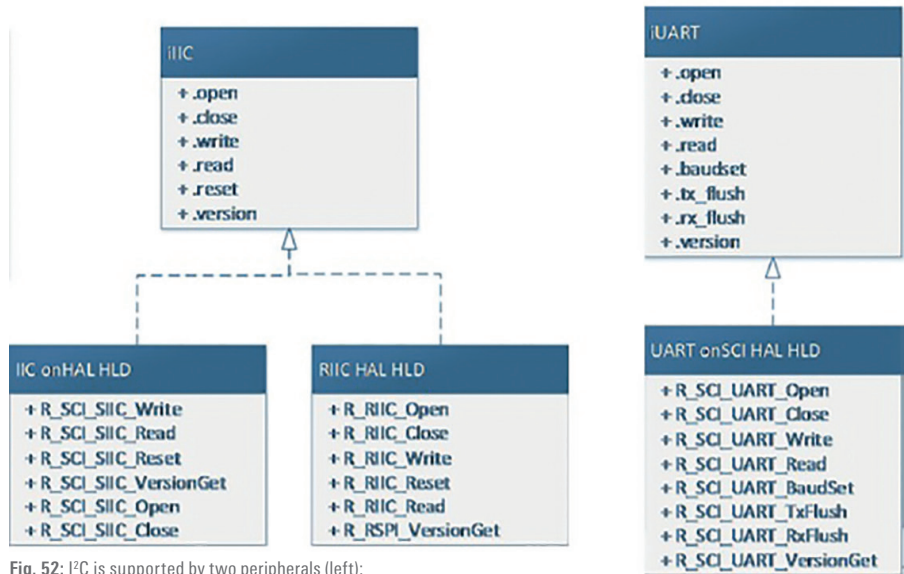


Fig. 52: I²C is supported by two peripherals (left); one-to-one mapping of UART (right)

As depicted in Fig. 51 the HAL is part of the Synergy Software Package. It contains many components reflecting the peripherals of the underlying hardware of the MCU (refer to chapter 2.3) including communication, analog modules, timers, and many more. This correspondence to the peripheral modules is also visible in the dedicated module names for simple identification. The HAL operates fully independent of any operating system that is also used.

The internal structure of each HAL component is split into two components, separating again the hardware related part from the interfacing part. The LLD (Low-Level-Driver or implementation module) has direct access to the hardware via the BSP and manipulates the peripheral registers directly. It uses versions of the same peripheral seamlessly. The HLD (High-Level-Driver or module interface) does not access registers directly, but is specific to the MCU hardware peripherals. The HLD provides the API to higher level modules and user code. As a common naming convention all module names begin with R_ for simplified identification, e.g. R_CGC for the Clock Generation Circuit.

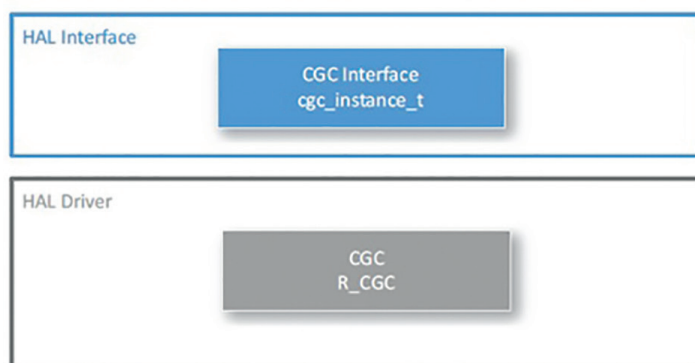


Fig. 53: Split of a HAL component (here Clock Generation Circuit) into LLD and HLD

The HAL of the SSP consists of 38 modules to support the hardware modules of the Synergy MCUs. Each module is well-described in detail in the SSP User's Manual. This description includes the description of the functions of both the HLD and the LLD.

As each HAL module that is used by an application has some memory requirements (remember: memory is a limited resource...) the memory requirements are listed in the SSP Datasheet – together with a short description of the functionality of the module.

r_cgc	Flash (Bytes)
S124 MCU Group	4,251
S3A7 MCU Group	4,127
S5D9 MCU Group	4,127
S7G2 MCU Group	4,127

Fig. 54: Memory usage for CGC using GCC compiler

How to add a module to a thread and how to configure the module to your needs is described in chapter 7 and chapter 12 in detail. But let's have a short look at the "Blinky" project that can be selected when setting up a new project (refer to chapter 12). The purpose of this small project is similar to the famous "Hello World" program – a simple starting point. The "Blinky" program will toggle the LEDs of the Starter Kit – nothing more, but even this short introduction shows some details of how to use the modules.

The "Blinky" project automatically adds and configures the needed HAL driver modules to the project like

```
- r_cgc - Clock Generation Circuit driver
- r_elc - Event Link Controller driver
- r_ioport - IO Port driver
```

Driver modules CGC and IOport are self-explanatory (setting up of IO-ports and clock tree), the Event Link Controller links different peripherals together using event requests.

The main() function just calls generated and user code (it will be regenerated by every click on the "Generate Project Content" button, so do not edit). The application code for the "Blinky" project is stored in blinky_thread_entry.c in the src directory, see Fig. 55. bsp_leds_t is a structure provided by the BSP API that contains information about the number of LEDs and the corresponding pins. ioport_level_t is a data type indication the level that can be set or read for individual pins, so either high or low. The first BSP API function that is used is R_BSP_LedsGet(&leds). This function returns information about the LEDs on the board. The first if-clause then checks whether any LED is available on the board whereas the second if-clause is used to toggle the LED state. Within the for loop the update of all LEDs is done by using the g_ioport as IO port driver instance. .pinWrite is a pointer to the pin write function to write the specified level (level) to the selected pin (leds.p_leds[i]). The last function tx_thread_sleep is provided by the RTOS ThreadX (chapter 7) to suspend the calling thread for the specified number of timer ticks (delay).

With just some functions provided by the APIs, the realization of this project is very simple. If you are unsure about how to use the functions or about the structures, just use the smart manual. Press "Ctrl+Space" at a variable or function and get all options and templates that are available like depicted in Fig. 56.

```

** File Name      : blinky_thread_entry.c
#include "blinky_thread.h"
void blinky_thread_entry(void)
{
    /* Define the units to be used with the threadx sleep function */
    const uint32_t threadx_tick_rate_Hz = 100;
    /* Set the blink frequency (must be <= threadx_tick_rate_Hz */
    const uint32_t freq_in_hz = 2;
    /* Calculate the delay in terms of the threadx tick rate */
    const uint32_t delay = threadx_tick_rate_Hz/freq_in_hz;
    /* LED type structure */
    bsp_leds_t leds;
    /* LED state variable */
    ioport_level_t level = IOPORT_LEVEL_HIGH;

    /* Get LED information for this board */
    R_BSP_LedsGet(&leds);

    /* If this board has no leds then trap here */
    if (0 == leds.led_count)
    {
        while(1); // There are no leds on this board
    }
    while (1)
    {
        /* Determine the next state of the LEDs */
        if(IOPORT_LEVEL_LOW == level)
        {
            level = IOPORT_LEVEL_HIGH;
        }
        else
        {
            level = IOPORT_LEVEL_LOW;
        }

        /* Update all board LEDs */
        for(uint32_t i = 0; i < leds.led_count; i++)
        {
            g_ioport.p_api->pinWrite(leds.p_leds[i], level);
        }

        /* Delay */
        tx_thread_sleep (delay);
    }
}

```

Fig. 55: Application code of "Blinky" project

```

25     /* LED state variable */
26     ioport_level_t level = IOPORT_LEVEL_HIGH;
27
28     /* Get LED information for this board */
29     R_BSP_LedsGet(&leds);
30
31     /* If this board has no leds then trap here */
32     if (0 == leds.led_count)
33     {
34         while(1); // There are no leds on this board
35     }
36
37     while (1)
38     {
39         /* Determine the next state of the LEDs */
40         if (IOPORT_LEVEL_LOW == level)
41         {
42             level = IOPORT_LEVEL_HIGH;
43         }
44         else
45         {
46             level = IOPORT_LEVEL_LOW;
47         }
48
49         /* Update all board LEDs */
50         for (uint32_t i = 0; i < leds.led_count; i++)
51         {
52             g_ioport.p_api->pinWrite(i, level);
53         }
54
55         /* Delay */
56         tx_thread_sleep (delay);
57     }
58 }

```

- pinCfg : ssp_err_t (*)(enum e_ioport_port_pin_t, unsigned long int)
- pinDirectionSet : ssp_err_t (*)(enum e_ioport_port_pin_t, enum e_ioport_dir)
- pinEthernetModeCfg : ssp_err_t (*)(enum e_ioport_eth_ch, enum e_ioport_eth_mode)
- pinEventInputRead : ssp_err_t (*)(enum e_ioport_port_pin_t, enum e_ioport_level *)
- pinEventOutputWrite : ssp_err_t (*)(enum e_ioport_port_pin_t, enum e_ioport_level)
- pinRead : ssp_err_t (*)(enum e_ioport_port_pin_t, enum e_ioport_level *)
- pinWrite : ssp_err_t (*)(enum e_ioport_port_pin_t, enum e_ioport_level)
- pinsCfg : ssp_err_t (*)(const st_ioport_cfg *)

Press 'Ctrl+Space' to show Template Proposals

Fig. 56: Application code of "Blinky" project

7. RTOS

Using the modular architecture and the APIs provided by the BSP and the HAL simplifies the programming and the interaction with the hardware significantly. But still there are some things to do to realize really good IoT applications. The available hardware resources like memory have to be managed and the software modules have to cooperate smoothly. You also have to cope with the increasing complexity of the application and the size of the code. And last but not least the timing requirements of the applications have to be met, in case of real-time systems in a deterministic way. According to market studies for embedded projects real-time capability is the key issue for most projects, followed by digital and analog signal processing and networking.

Of course we can just write the code for the application from scratch using the BSP, HAL or even higher layers like middleware (chapter 9 and 10) or functional libraries (chapter 8). But for sure it will be difficult or even impossible to fulfil all requirements by all means, in particular any kind of real-time requirements. So, some more support to manage the requirements like timing behaviour and resource management in a simple and reusable manner is highly appreciated.

This support will be given in general by an operating system, for any application with real-time requirements it will be an RTOS (Real-Time Operating System). RTOS are used in more than 2/3 of current embedded projects and the usage of an RTOS is more or less a must-have. If an RTOS is used, in most cases a commercial RTOS is commonly used instead of any proprietary solution. Let's have a look at real-time operating systems starting at the second part – the operating system.

As we have already seen in the previous chapters the resources of a microcontroller (as well as of all other digital units) are more or less limited. For a MCU there is maybe just one CPU (like for the Synergy MCUs, there are also MCUs with 2 or more CPUs), the memory size is limited and there are some peripherals. Therefore these resources have to be managed somehow. Of course this management can be done without an OS, but an OS provides all the features you need – so just use it.

According to the German standard DIN 44300 an operating system (OS) is a bundle of software programs to support application programs:

- operate the digital hardware
- manage and allocate the hardware resources like CPU and memory
- provide common services for other software parts
- manage and control the execution of the software

By this support of the OS the hardware is made usable for the user. The OS tracks the status of the current user of each resource and schedules this resource, e.g. the CPU. Consider several programs or tasks should run on a MCU with just one CPU (Fig. 57). The execution of these concurrent tasks has to be managed as real concurrency is of course not possible with just one CPU. If the multitasking of the tasks is done properly the OS can pretend the concurrent execution of the tasks. In addition the OS takes care of and resolves conflicts for resources and optimizes the performance.

To make the services available for the user the OS provides an API and the user code can directly use the API functions. The interface towards the hardware is again an HAL.

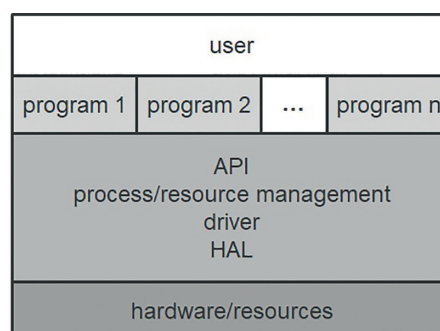


Fig. 57: Simple schematic of an OS layer model

There are many, many operating systems available, some are suitable for embedded systems, some are not suitable depending on the characteristics of the OS:

- Abstraction of the hardware to simplify life for the user
- Dedicated to the underlying hardware architecture and structure
- Efficient use of the hardware
- The OS itself requires some system resources like memory size
- CPU load for the execution of the OS

Correlating these requirements with the hardware resources of MCU it becomes obvious that the OS has to fit to the hardware. Commonly known and used operating systems for PCs like Windows are not suitable for embedded systems as they require a huge amount of memory – which is not available on a MCU (some MB memory, not GB...). So for embedded systems a small memory footprint of the OS is mandatory.

Now let's move to the first part of RTOS – the real-time. What does real-time mean for embedded systems? The key topic of real-time is the deterministic timing behaviour in terms of well-defined response in time to internal or external events. After an event input signals are processed and output signals are generated in general. Well-defined response means that both the functional and timing response is correct – one without the other is useless. And both functional and timing response depend on the application. Functional is obvious, but what about the timing? It does not necessarily mean fast, but defined by the system. For example the response time of a heating system of a house can be rather slow – nobody will notice whether the timing is in millisecond or second range when setting a new temperature. On the other hand an emergency braking system of a car needs very fast response times, just a few milliseconds from obstacle detection to brake actuation. Some seconds will be much too slow...

Taking the example of the emergency braking system we can find the basic requirements for real-time systems. The braking system has to fulfil strict timing requirements, it has to be always available (at least when driving) and it has to share the hardware resources of the ECU (Electronic Control Unit) with other applications. Hence the three basic requirements for real-time systems are:

- Timeliness
- Availability
- Concurrency

Timeliness is the key element of any real-time system and its key feature is the deterministic behaviour. The real-time system reacts to events and generates some kind of result or output and the complete response time of the system to an event has to fit to the requirements of the application, always and without any exception. The events for triggering the actions of the real-time system can be external as well as internal, and these events can happen occasionally, periodically or irregularly. Depending on the application the measure for the response time can be absolute (e.g. every 10 seconds) or relative (e.g. 10 ms after the event occurred).

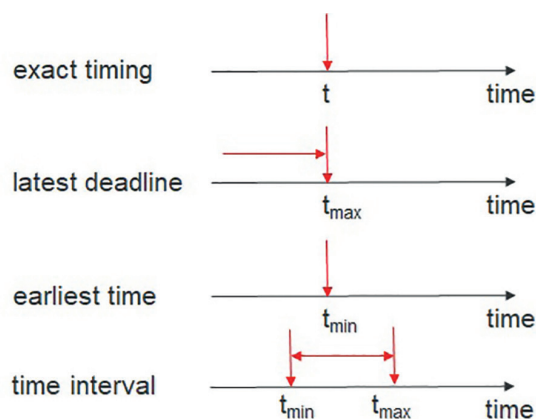


Fig. 58: Types of timeliness

As the application determines the timing requirements the system has to fulfil it also determines how strict these requirements are. And some exceptions are maybe tolerable for some applications and systems. Therefore we can distinguish between two types of real-time, soft and hard real-time.

Hard real-time describes exactly what we defined so far. Hard real-time applications have highly critical time constraints with a fixed deadline. Meeting the timelines is essential and has to happen always without any exception. Missing any event is unacceptable as it could result in a catastrophic failure of the application. Fig. 59 depicts the value of the result generated by a hard real-time system with the latest deadline requirement on the left side.

Until the deadline the value of the result is as high as possible. The system response is correct if also the functional result is correct. After the deadline the result is worthless, even in case the functional result is correct. The system cannot use the result any more as it is too late. Just reconsider the example of the emergency braking: the system detects an obstacle and (external event) and has to calculate whether to brake or not, depending on the speed of the car. If braking action has to happen the brakes have to be actuated within a defined time. If the functional correct output is generated and the timeline is met, the car stops in front of the obstacle. If the timeline is not met, there will be an accident not matter of the correct functional behaviour.

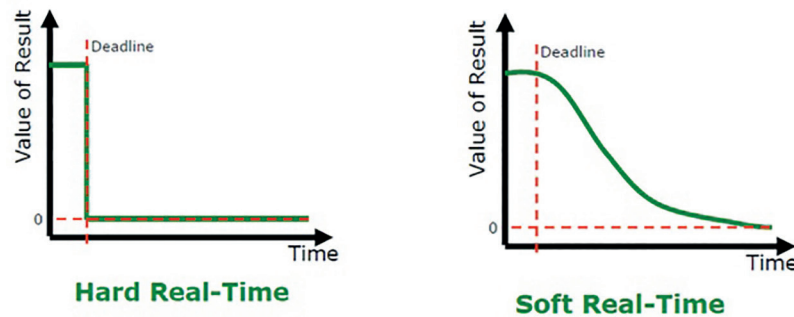


Fig. 59: Types of real-time: hard real-time (left) and soft real-time (right)

Situation is different for soft real-time applications. Here again of course the functional behaviour has to be correct. But this time the time constraints are less critical and deadline is somehow soft. The timings should be kept in general, but if sometimes the deadlines are not met it can be accepted to some extent. Missing an event is undesired but not critical and does not result in a catastrophic behaviour. Consider an application everyone knows, a keyboard, as a typical example for a soft real-time behaviour. Target of the system is that any character you type on the keyboard appears immediately on the screen. Let's say within 100 ms. As there are many other programs running on the PC the transfer from keyboard to display might be delayed (see also section about concurrency below). If it is 200 ms sometimes, nobody will care, if it's a second, you will start to get angry and any longer time will be unacceptable. But still you are able to work, just slower... This behaviour is depicted in Fig. 59 on the right side. Until the deadline the value of the result is maximal. After the deadline the value decreases but still is useful to some extent.

As we have seen real-time systems have to generate the correct response to events in time any time. Hence the system has to be available all the time as events can occur at any time. An interruption of operation is not allowed by no means. Something like the famous Windows blue screen must not happen... This availability is the second prerequisite for real-time systems. To ensure the availability some measures have to be taken for the system including the MCU and the software. First of all you need an uninterruptable power supply. The MCU has to be always in a mode to be able to react on dedicated events in time. This is in particular important in cases when the MCU is in a low power mode (see chapter 2) and not running in full speed. And the software to serve the event must not be blocked by any other running task, or at least it has to be executed in time.

Third requirement for real-time systems is the concurrency of several tasks that should run in parallel. Have a look at yourself – are you able to multitask? If yes, how does it work with your single CPU called brain? In fact the brain does one task at a time, then changing to another task. If the change of tasks is done very fast, it seems like the tasks are executed in parallel. Same holds true for real-time systems (at least as long as the number of CPUs is smaller than the number of tasks to run in parallel). In general the real-time system has to handle several tasks (called threads later on). Each of these tasks has some deadline according and each of the deadline has to be met taking the other tasks into account. A really parallel execution of the tasks is impossible due to the limited resources of the system (one of some few CPUs). Therefore the tasks have to be executed quasi-parallel to achieve multitasking. Quasi-parallel means a sequential execution of the tasks meeting all timing requirements by proper scheduling and prioritisation of the tasks. So at any moment it has to be decided which tasks gets the CPU for execution. The scheduling of tasks plays a major role for any real-time system and real-time operating system.

Finally an RTOS is an operating system with additional features for real-time operation. These features include

- Deterministic timing behaviour
- Fast operation (at least in general, depends...)
- Short latency
- High number of external and internal events
- Limited resources like CPU, memory, ...
- Small overhead, e.g. for process change, ...
- Reliability
- Control and scheduling of many concurrent tasks
- Prioritization of tasks

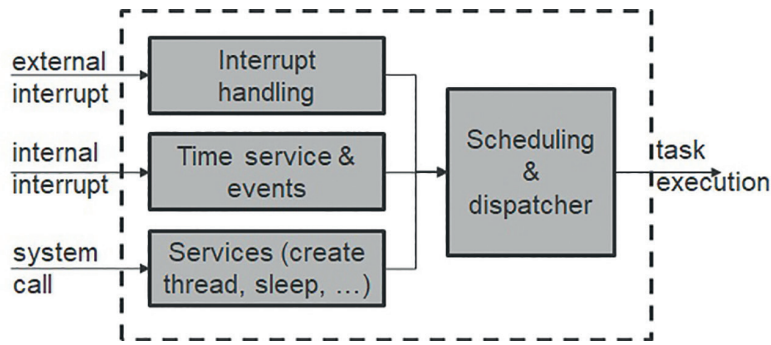


Fig. 60: Simple schematic of an RTOS

Why is a “normal” OS like Windows not suited for real-time embedded applications? Besides the large memory space it provides too many features that are not needed for embedded applications and a large overhead that consumes the limited resources as it is not designed for embedded systems. It is also not designed for mission-critical and real-time applications as both reliability (blue screen) and unpredictable timing behaviour run contrary to the three real-time requirements.

The theory and concepts of RTOS are manifold and there are many excellent books about RTOS (see references at the end of this book) so just a short introduction to some basic concepts will follow. The core of any RTOS is the kernel. It provides the services to control the hardware like CPU and memory. It also provides the services for the embedded applications like I/O management, interrupt and event handling, thread and timer management or communication. It is the first program loaded into memory on start-up. To fit to the limited resources its memory size is as small as possible.

One of the key functions of an RTOS is the management of the different tasks or threads of a software. The definition of tasks and threads is sometimes a bit confusing and both terms are sometimes used exchangeable. To be compliant with the wording of ThreadX® (ThreadX® is the RTOS of the SSP) let’s clearly distinguish these two terms and later on just the term thread will be used.

A program has to execute several tasks, in general concurrently. A thread is a semi-independent segment of the program and the smallest sequence of instructions that can be managed independently by a scheduler. In general an application will have a separate thread for each distinct activity. All threads together form the program. As the purpose of some threads is more important than others a priority is assigned to the threads. The higher the priority, the higher the importance of the thread. The threads are executed concurrently and share the same memory space within a program. To realize proper concurrency some kind of scheduler is needed to allocate the hardware resources like the CPU to a thread with a defined priority. How the scheduling is done depends on the architecture of the RTOS.

To enable multitasking and scheduling threads can be in different states like running or suspended. Each thread is in one of these states and just one of the threads is in executing state. For ThreadX® there are five states with corresponding state transitions (Fig. 61 and Table 5). At any moment just one thread is in executing state controlling the processor. If a thread with higher priority becomes ready, the execution of the current thread is interrupted (Fig. 62). This thread returns to ready state and the new thread with higher priority enters executing state.

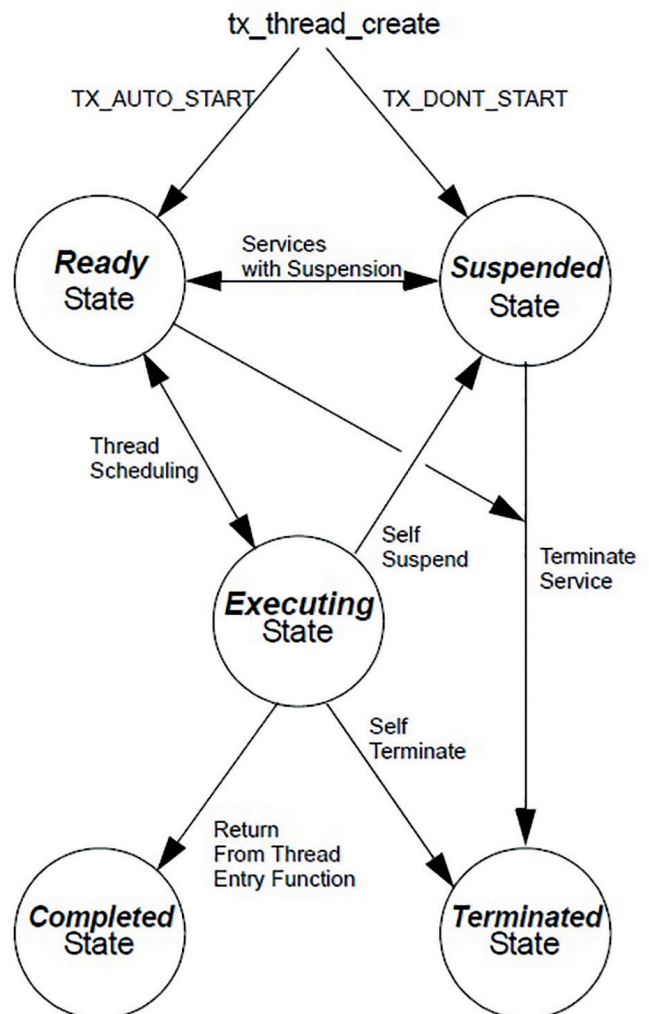


Fig. 61: ThreadX® states and state transitions

State	Characteristics
Ready	<ul style="list-style-type: none"> - Thread is ready for execution - Transition to executing state as soon as it is the highest-priority thread in ready state - Comeback to ready state from executing state in case of occurrence of a higher-priority thread
Suspended	<ul style="list-style-type: none"> - Thread not eligible for execution - Waiting for semaphores, mutex, time, ... - Explicit state suspension
Executing	<ul style="list-style-type: none"> - Thread is running - Only one thread in executing state at any moment - Control of processor
Terminated	<ul style="list-style-type: none"> - Thread terminated by itself or by another thread - Needs reset to original state before next execution
Completed	<ul style="list-style-type: none"> - Thread has completed - Needs reset to original state before next execution

Table 5: ThreadX® states

To be able to react on an internal or external event microcontrollers provide interrupts. The RTOS can use these interrupts to control the CPU usage. As already introduced in chapter 2 interrupts are hardware or software signals to the processor that an event has occurred. Events can be generated by a signal change of a pin, a timer module or software for example. The action for this interrupt is handled by an ISR (Interrupt Service Routine, Fig. 14 and Fig. 62). An ISR is dedicated to an interrupt and contains the code that handles the actions needed to serve the interrupt. All ISRs are stored in a dedicated memory location and an interrupt vector containing the starting address of the ISR is associated with each interrupt. As interrupts are asynchronous in general they may occur at the same time or an interrupt can occur during the execution of a thread or another ISR. If an interrupt occurs during thread execution the thread execution is stopped and the ISR is executed. In addition the importance of interrupts is different and hence a mechanism is needed to deal with the different importance levels.

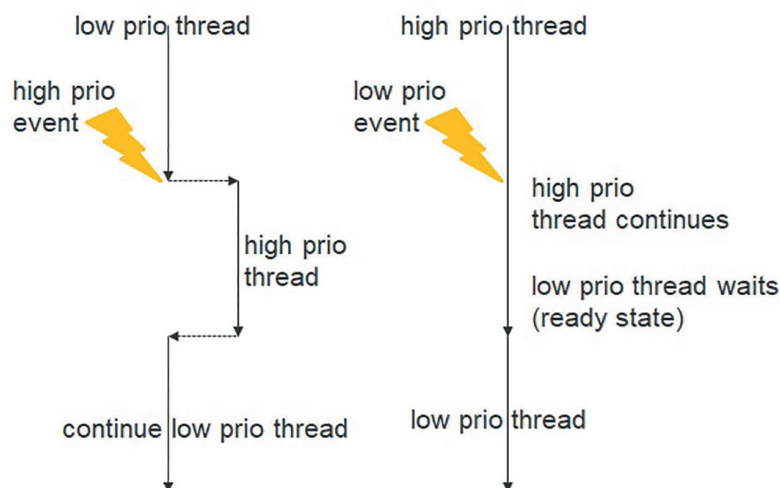


Fig. 62: Thread and interrupt priorities respectively

The concept of interrupt priorities is used to distinguish the importance of interrupts and control the order of execution of the ISRs associated with the interrupts. This priority mechanism is depicted in Fig. 62 (just read ISR instead of thread).

Consider a running ISR (left of Fig. 62). If an interrupt with a priority higher than the priority of the running ISR occurs, the running ISR is interrupted and switches to ready state. The current status of the processor including program counter, register values is saved to the stack during the context switch and the high priority ISR starts in executing state. After this ISR is finished the processor status of the former ISR is restored and the first ISR continues operation in executing state.

No interruption of the running ISR will happen if the second interrupt has a lower priority. In this case the first ISR stays in executing state and the second one enters ready state. After the first one is finished it releases the CPU and the second one can enter executing state.

If the assignment of priorities to the interrupts is done properly this mechanism is very well suited for real-time applications as the order of execution is determined by the priority level and can be adapted to the deadlines of the ISR.

There are several different scheduling algorithms for RTOS like control loop with polling, round robin or earliest due date. The method of scheduling the CPU resource by prioritization of threads and ISR is called preemptive scheduling. It provides an excellent way of prioritizing your threads and interrupts and to act fast and with a deterministic timing behaviour. It fulfils the three requirements of real-time systems, timeliness, availability and concurrency. The importance of the thread and ISR respectively is taken into account by suitable priority levels. If several threads (or ISR) have to be processed the one with the highest priority is executed.

Some essential characteristics for the pre-emptive scheduling:

- Running thread can be interrupted and suspended by a higher priority thread
- Status of suspended thread is saved (context switching)
- Resume of execution of suspended thread after interrupting thread is finished

A simple example for pre-emptive scheduling is the following doorbell example. You are at home and just idle away. There are three asynchronous interrupt sources to wake you up with corresponding actions you start (your ISR): the doorbell, a burglar alarm and a fire alarm. Obviously these three actions have different importance: the doorbell has lowest importance and the fire alarm highest (if you have other priorities, please feel free to change...). Correspondingly the deadline for the fire alarm is the shortest and for the doorbell it is longer. Without prioritization of the interrupts and pre-emptive scheduling you might lose your house. If you are currently answering the doorbell and suddenly the fire alarm starts, you will take the time to finish your doorbell action before reacting on the fire alarm. Unfortunately it's too late...

With pre-emptive scheduling you will keep your house: The doorbell rings and the action for the doorbell starts execution. The burglar alarm interrupts your doorbell action and as the alarm has higher priority, you start searching the burglar and suspend your doorbell action (keeping in mind what you were doing at that moment). During your search the fire alarm rings. You keep in mind that you were searching a burglar and you start to fight the fire. After the fire is extinguished you continue to search the burglar. You return to the doorbell right after you expelled the burglar.

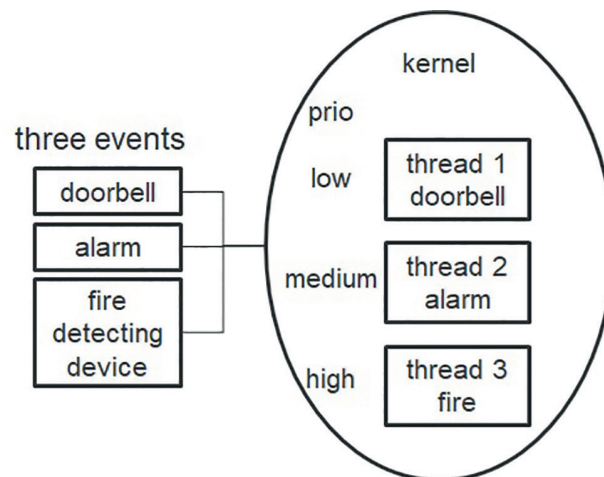


Fig. 63: Pre-emptive scheduling example with three interrupt sources and prioritization

The concept of pre-emptive scheduling provides many benefits. Threads can be prioritized and the responsiveness to events is very high. The behaviour is deterministic and the scheduler of the RTOS controls when to execute or interrupt each thread. Hence the developer does not have to take care about scheduling (just about prioritization) and it is possible to develop the threads independent of each other. But there are also some issues you have to take care of with pre-emptive scheduling and synchronization of threads.

One issue is the synchronization of threads with regard to shared resources like a memory. A thread may need exclusive access to a shared resource. Even if this thread is interrupted, an access to the shared resource by the interrupting thread has to be prevented to avoid any malfunction of the system.

In a simple example there are two threads: one is writing data into a memory area, the second one with higher priority reads the data from memory. Obviously both threads should not access the memory at the same time or interleaving. Therefore the write thread has to lock the access to the memory. If the access is locked and the read thread interrupts the first thread, it cannot access the memory and it has to wait until the first thread releases the lock. This locking is called mutual exclusion or mutex. It is a mutually exclusive flag. Just one thread can have the key to a shared resource and proceed with its work. Even if it is interrupted by a higher priority thread, the interrupting thread has to wait if it requires access to the locked resource. The first one continues, releases the mutex, and now the higher priority thread can continue with its access to the shared resource.

The concept of mutual exclusion can be generalized by the concept of semaphores. A semaphore is a signalling mechanism. It restricts the number of simultaneous users of a shared resource up to a maximum number. Threads can request access to the resource (decrementing the semaphore), and can signal that they have finished using the resource (incrementing the semaphore).

Operating systems like ThreadX® provide both mechanisms in general to control the access to shared resources.

Another issue is thread starvation when threads with a low priority are not executed any more. This low thread does not execute until there are no higher priority threads ready for execution. If a higher priority thread is always ready, the lower priority thread will be blocked. Here the developer has to take care to avoid or solve this issue. A careful application design with proper use of priorities might solve the thread starvation. Also take care that higher priority threads don't execute continuously blocking lower priority threads. Or add some algorithm to the application to raise the priority of the starved thread gradually until it gets executed.



Fig. 64: Thread starvation of the low priority thread

Shared resources can also lead to the problem of priority inversion. It is a nondeterministic timing behaviour (that's really very bad for a deterministic RTOS) and describes the situation that a lower priority thread finished earlier than a higher priority thread. Let's assume a system with three threads of low (L), middle (M) and high (H) priority respectively. L and H share a common resource like a memory and they use a mutex to control the access to this resource. M does not share this common resource. As M and H do not share a common resource H always has to finish earlier than M.

Now L is executed and locks the resource by a mutex. Now H interrupts L and starts execution until it needs access to the shared resource. As it is locked by L H waits for L to continue and release the resource. Now M also interrupts L and starts running until it is finished as it does not access the resource. L continues until it releases the mutex, then H can continue and finish, afterwards L can end its execution. But this time M finished earlier than H even though they do not share a common resource – the priority of these two threads is inverted. In an RTOS this can result in severe outcomes.

Priority inversion can be handled through careful selection of thread priorities or by using methods of the RTOS like pre-emption-threshold and pre-emption inheritance. Pre-emption threshold allows a thread to specify a priority ceiling for disabling pre-emption. Threads that have higher priorities than the ceiling are still allowed to pre-empt, while those less than the ceiling are not allowed to preempt. In our example the ceiling can be set in a way that H may interrupt L and M may not. When priority inheritance is used a lower priority thread can temporarily assume the priority of a high priority thread that is waiting for a mutex owned by the lower priority thread. This capability helps the application to avoid undeterministic priority inversion by eliminating pre-emption of intermediate thread priorities.

The most famous problem with priority inversion happened during the Mars Pathfinder mission in 1996/1997. Here VxWorks RTOS with pre-emptive scheduling reset the computer due to a priority inversion failure. It was fixed by a software update including priority inheritance.

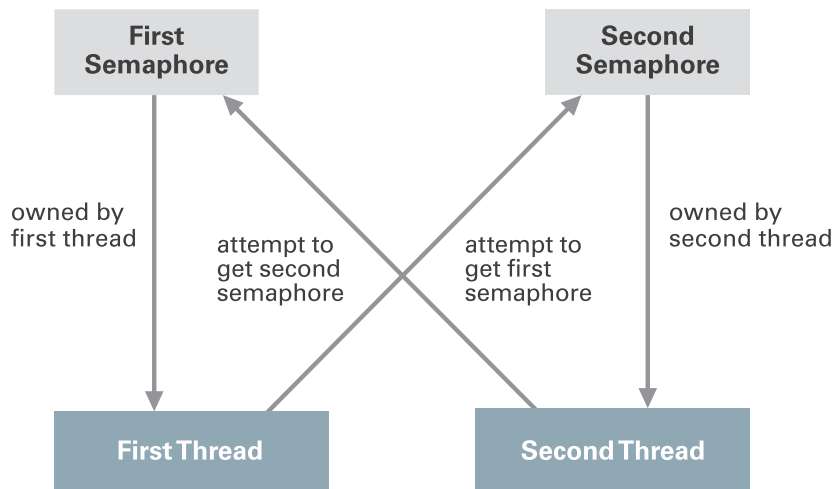


Fig. 65: Deadlock

Also deadly embrace or deadlock is caused by synchronization mechanism like a semaphore. During the deadlock two (or more) threads are suspended indefinitely while attempting to get semaphores that are allocated by the other thread. Like depicted in Fig. 65 both threads own a semaphore. If the first thread attempts to get the second semaphore and the second thread the first semaphore, both threads block each other. Deadly embrace can be avoided by placing restrictions on how semaphores are obtained by threads.

7.1. SSP RTOS

As the usage of an RTOS is very common for all embedded real-time applications the Renesas SSP already contains an RTOS. ThreadX® by Express Logic is an industrial proven multitasking RTOS for deeply embedded, real-time and IoT applications. It is MISRA-c:2004 and MISRA-C:2012 compliant and commonly used worldwide with more than 5.4 Billion deployments in applications like consumer devices, industrial control equipment or medical devices. For simple use it provides an intuitive and functional API. Due to its real-time scheduling algorithms and efficient multitasking it fulfils all requirements for real-time applications, in particular the strict deterministic behaviour. These features include round-robin scheduling, time-slice pre-emptive, priority-based and Preemption-Threshold™ scheduling and event-Chaining™ technology. To be able to react very fast to events the interrupt response time is very short and the context switching is fast.

ThreadX® supports the features you need to set up your real-time application. Besides the deterministic behaviour of course execution speed is important and ThreadX® combines a deterministic and fast execution of the code. Fig. 67 depicts some performance features like the optimized interrupt processing and context switching.

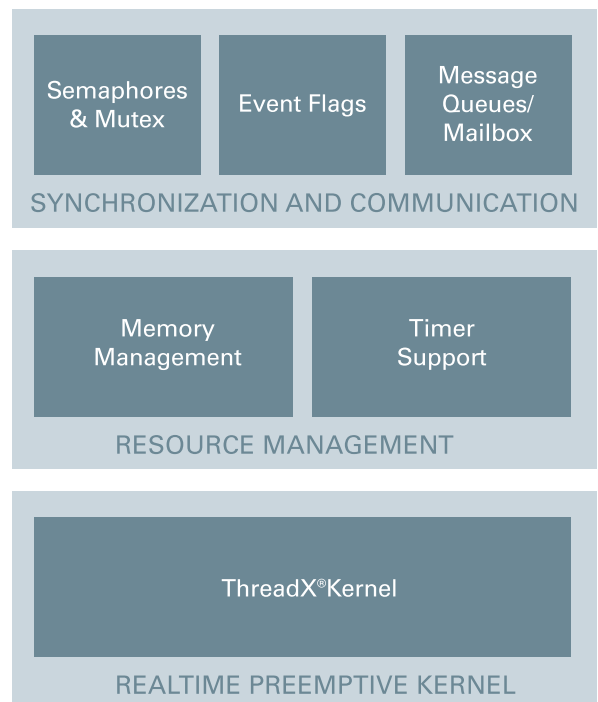


Fig. 66: ThreadX® features

SERVICE	TIME
Thread Suspend	0.6 μ s
Thread Resume	0.6 μ s
Queue Send	0.3 μ s
Queue Receive	0.3 μ s
Get Semaphore	0.2 μ s
Put Semaphore	0.2 μ s
Context Switch	0.4 μ s
Interrupt Response	0.0 μ s – 0.6 μ s

Fig. 67: ThreadX® performance figures for a processor running at 200 MHz

The management of the limited hardware resources is done in an efficient, fast and reliable manner:

- Several memory allocation methods like fixed-size memory blocks and multiple byte pools (similar to standard C heap)
- Application timers
- Synchronization of threads and communication
- Semaphores, mutex and methods like priority inheritance
- Event flags
- Message queues

Due to its Picokernel™ architecture it has a small memory footprint. Only services really used by the application are included in the final code to preserve the limited memory size as much as possible.

Last but not least ThreadX® is pre-certified for many standards. Remember many IoT applications have special requirements, e.g. with regard to safety. To fulfil these requirements the hardware and software has to be certified – and the complete system in the end. If the software has already proven that it is compliant with the required standards (like functional safety) it is a big advantage for your system, as you can rely on this pre-certification.

Standard	Description
IEC 61508 up to SIL 4	Functional safety of electronic safety-related systems
IEC 62304 up to SW safety Class C	Medical device software
UL 60730-1 H, CSA E60730-1 H, IEC 60730-1 H	Automatic electrical controls
UL 60335-1 R, IEC 60335-1 R	Safety of household and similar appliances
UL 1998	Standard for software in programmable components

Table 6: Certifications of ThreadX®

Using ThreadX® within e²studio is rather simple, as it is already an inherent part of the SSP and directly available after starting e²studio without any further effort. Like the other configurations within e²studio also the configuration of a thread and of the associated drivers and modules is done graphically – intuitive, reliable and fast. Even though you will practice during the lab, let’s have a look at this procedure taking the configuration of SW4 of the S7G2 Starter Kit to toggle a LED as an example. If SW4 is pressed, an interrupt shall be generated.

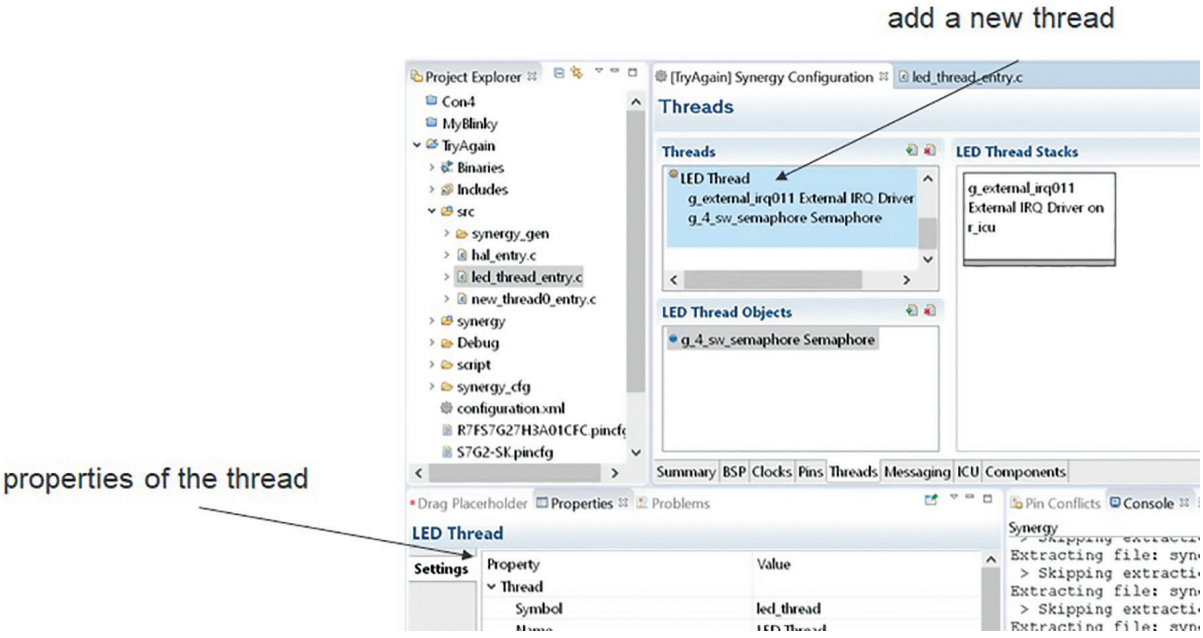


Fig. 68: Thread tab in e²studio’s configuration perspective

Fig. 68 depicts the configuration perspective of e²studio with the threads tab active. In the small Threads window all threads are listed. Adding a new thread is easily done by clicking on the small green plus sign in the Threads window. The properties of the thread are displayed in the left corner window. In the window next to the Threads window the drivers of this thread are displayed. Here it is the External IRQ Driver on r_icu that we will use to signal a falling edge on pin P006 that is connected to SW4.

The properties of the driver are displayed in the properties window as soon as the driver is selected by clicking on it (Fig. 69). Here all properties can be set according to the requirements of the application. In our example we connect SW4 to IRQ11 and use the falling edge of the switch to generate the interrupt. To prevent bouncing of the input a digital filter is switched on. The interrupt priority is set to a medium level and the corresponding ISR is assigned.

g_external_irq011 External IRQ Driver on r_icu

properties of the driver

	Property	Value
	Settings	
	Information	
	v Common	
	Parameter Checking	Default (BSP)
	v Module g_external_irq011 External IRQ D	
SW4 connected to IRQ11	Name	g_external_irq011
Catch pressing of SW4	Channel	11
Enable digital filtering to debounce the button	Trigger	Falling
	Digital Filtering	Enabled
	Digital Filtering Sample Clock (Only val	PCLK / 64
Callback function: ISR	Interrupt enabled after initialization	True
	Callback	external_irq11_callback
Priority of the interrupt	Interrupt Priority	Priority 8 (CM4: valid, CM0+: invalid)

Fig. 69: Properties of the External IRQ Driver

Last step of the configuration is the assignment of pin P006 to the IRQ11 which is done in the pin configuration tab (Fig. 70).

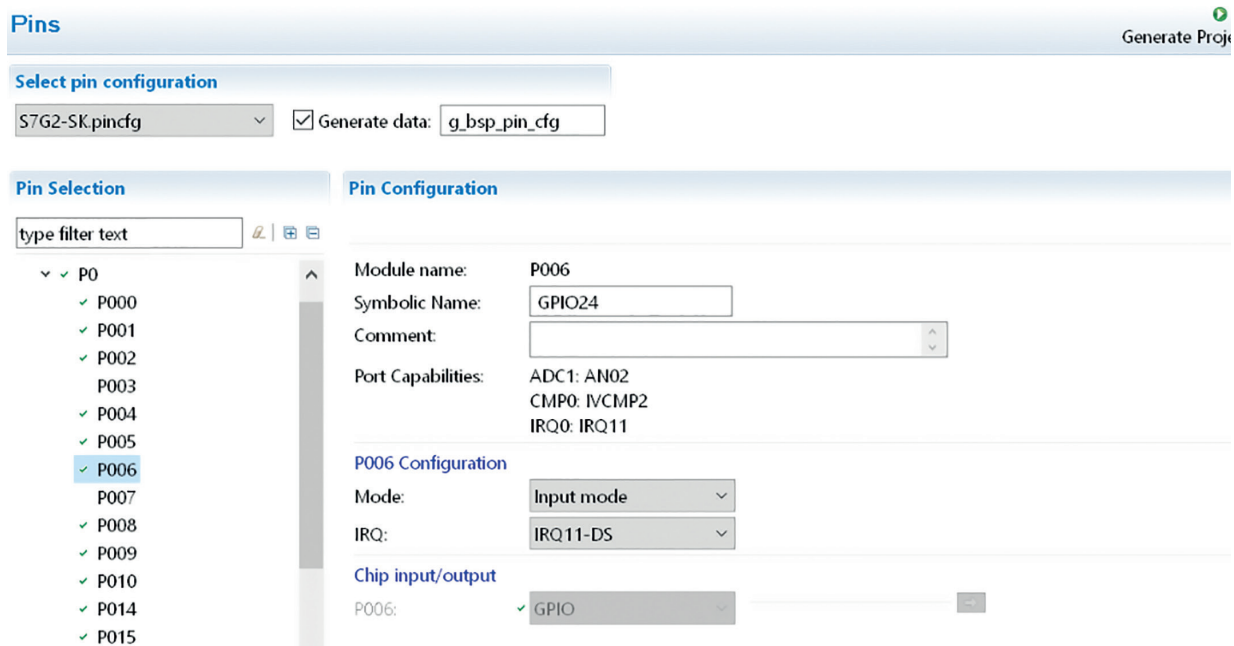


Fig. 70: Assignment of P006 to IRQ11 in pin configuration tab

The code is of course generated automatically after clicking the “Generate Project Content” button.

Using an RTOS makes the development of real-time applications much simpler and more reliable compared to applications without an RTOS. But still the behaviour of the system has to be tested, in particular the timing behaviour. Hence a major role during development of real-time applications (and of all kind of applications) is the analysis of the functionality, timing and performance of the application and the debugging. To support these tasks the use of an adequate tool is highly recommended. Express Logic provides a host-based analysis tool that integrates seamlessly with the Renesas SSP. TraceX® has the features to analyse and optimize the real-time applications that are not available with standard debugging tools:

- Graphical view of real-time events like interrupts, context switches and semaphores
- Analyse and solve problems
- System fine-tuning
- Optimization of performance and efficiency

The graphical view can be used to check the order of thread execution and thread execution profiles, analyse delta ticks between events or the stack usage and to find real-time issues like priority inversion. The system performance can be analysed as well. If middleware by Express Logic like NetX™ and FileX® is used in the application the tool can also be used to generate statistics for this middleware.

TraceX® is part of the Renesas Synergy Platform and is available for separate download from the Synergy Gallery.

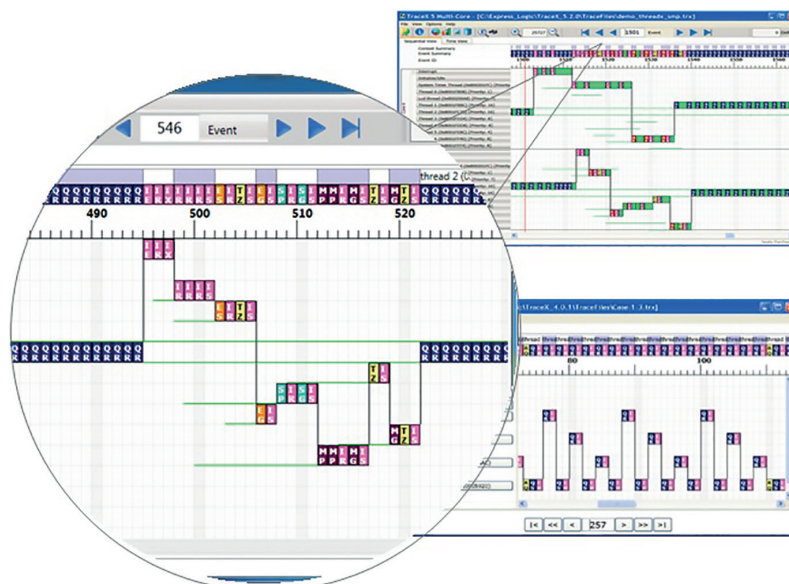


Fig. 71: Screenshot from TraceX®

8. APPLICATION FRAMEWORK & FUNCTIONAL LIBRARIES

The RTOS provides everything that is needed to realize real-time applications and both the BSP and the HAL provide the low level support we need to simplify the interaction with the hardware by using the corresponding APIs. But still this level of abstraction is rather limited. Complex applications can be built up by a suitable combination of HAL modules. This building process needs the detailed knowledge of the modules. Therefore it would be better to have predefined parts that already incorporate this detailed knowledge and provide a higher level of abstraction to focus on application, not implementation.

Robots, not only in production plants but in other applications as well, are getting more and more popular. In addition, the degree of autonomy of the robots and of systems in general is increasing very fast and autonomous systems operate without any human interaction. Consider an industrial or IoT application with all the components you have to connect to your MCU, e.g. sensors, actuators, connectivity, HMI and much more. Programming at HAL level is of course possible – but it will be difficult and error-prone and takes a lot of efforts. An automated robot in a production plant for example has several requirements for the embedded control system, in particular it is cooperating with an operator. Sensors are needed for positioning and speed measurements, to detect obstacles like the operator or to measure temperature, pressure or any other required parameter. These sensors interface with the MCU via analog or digital interfaces like I²C or SPI. The embedded system controls the actuators on the other hand, e.g. electric drives or valves. For this purpose analog and digital outputs are needed, for electric drives PWM generation (Pulse Width Generation) is a must. To calculate the control algorithms based on the sensor inputs and the user control digital signal processing has to be done on the MCU. To set up digital signal processing from scratch just using the HAL requires a deep knowledge of the algorithms as well as deep knowledge of the HAL. Not to mention sophisticated HMIs like a touch screen. Powerful higher level modules are highly appreciated to support the application development without the need to consider the underlying hardware and low level software.

Recap the ideas of modular software. One basic idea was the combination of software modules to form more complex functions. Like depicted in Fig. 72 modules can be stacked to realize new and complex functions. The stacking is rather easy as the modules have defined interfaces. They can provide services to other modules or require service from other modules using these interfaces.

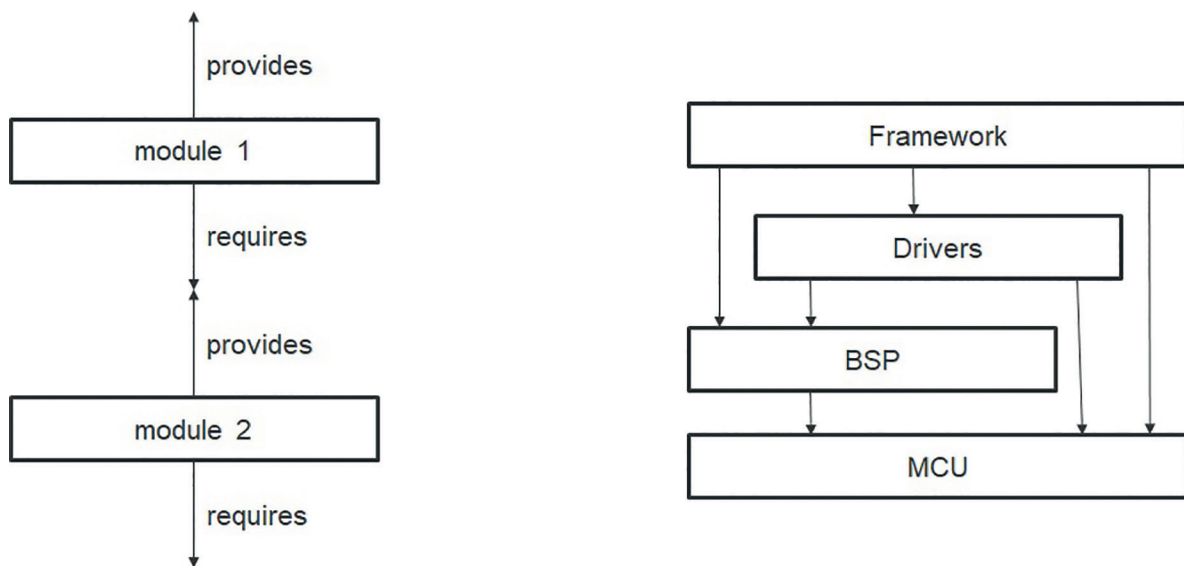


Fig. 72: Schematic of a module stack (left); example of a framework stack (right)

Application frameworks, functional libraries or so called middleware (refer to chapters 9 and 10) realize the required abstraction. They provide system level services frequently used in user applications. Hence you don't have to reinvent these commonly used tasks again and again, just reuse the framework. An application framework is in general a stack of modules with corresponding APIs. The framework layers can use HAL modules, the BSP or even the MCU directly (Fig. 72 right). By using a specific framework API stack underlying software stacks can be exchanged without having to adjust the user code.

Frameworks usually do not depend on a peripheral or a specific driver. Instead the underlying hardware can be changed without changing the user code. The console framework can be used as an example: The console application can be implemented by a UART, USB or Telnet interface. For the application it is not important which stack is used as it just uses the top level API of the console framework. Console framework services and features stay the same (sf_comms) but the lower layer implementation may differ as depicted in Fig. 73. This feature enhances flexibility, scalability and portability significantly.

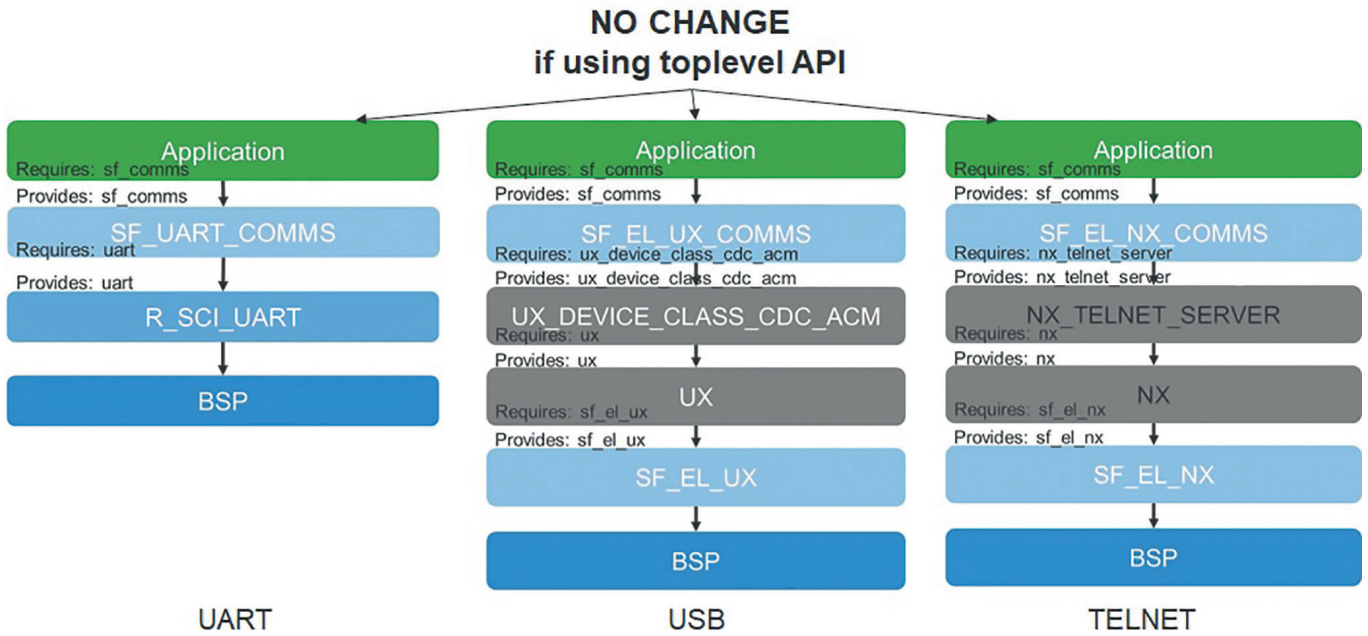


Fig. 73: Console framework with three different implementations

By using application frameworks you neither have to care about the underlying hardware, nor you have to be an expert on Renesas' MCUs. Just use the functions of the API of the framework. Besides the reduced complexity and a high level of abstraction the usage of application frameworks has many other benefits:

- Increase of portability
- High degree of flexibility
- Reduced low level programming
- Higher efficiency, reliability, faster development
- Reuse of code across processors and products
- Reduced development time and shorter time to market
- Reduced cost
- Higher efficiency
- Longer lifetime of code - even when the hardware evolves (remember Moore's law...) the code survives
- Tested and verified code
- More consistent and reliable code

Besides the application framework functional libraries can also add additional capabilities and features. These libraries provide dedicated functionality often needed for embedded systems. Examples from SSP include DSP (Digital Signal Processing), cryptography and security libraries.

8.1. SSP APPLICATION FRAMEWORK AND FUNCTIONAL LIBRARIES

The SSP provides a large number of application frameworks and functional libraries commonly used in industrial and IoT applications including ADC, audio or GUI applications. The module name start with sf_. Frameworks are directly integrated into e²studio, so no additional effort is needed – just use them.

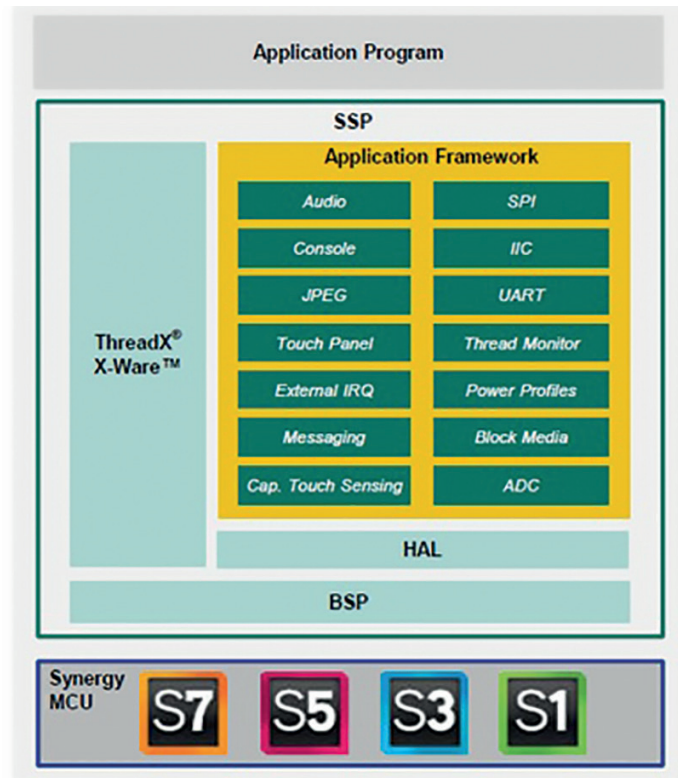


Fig. 74: Application frameworks of the SSP

The ADC framework serves as an example for application frameworks. It is used to sample and convert analog signals into digital data using the ADC channels provided by the MCU. The configuration of the ADC framework is very flexible. As depicted in Fig. 75 the ADC framework is located on top of the HAL and it uses several HAL modules, e.g.

- ADC for data conversion
- GPT (General Purpose Timer) for timing functions
- DMA/DTC for efficient data transfer

Getting the ADC framework up and running is again rather simple in e²studio. In configuration perspective select a thread that should include the ADC framework. Add a new stack to the thread by selecting Framework->Analog->ADC Periodic Framework on sf_adc_periodic. The used HAL modules are directly integrated and displayed in the stack window, ADC, GPT and DTC. Now the configuration can be done simply in the corresponding properties tab. In the framework toplevel configurations like data buffer size, sampling iterations or GPT trigger channel are set. The HAL module are also configured according to the application needs. E.g. 8-bit resolution, channels to scan and scan mode for the ADC or period and interrupt priority for the GPT.

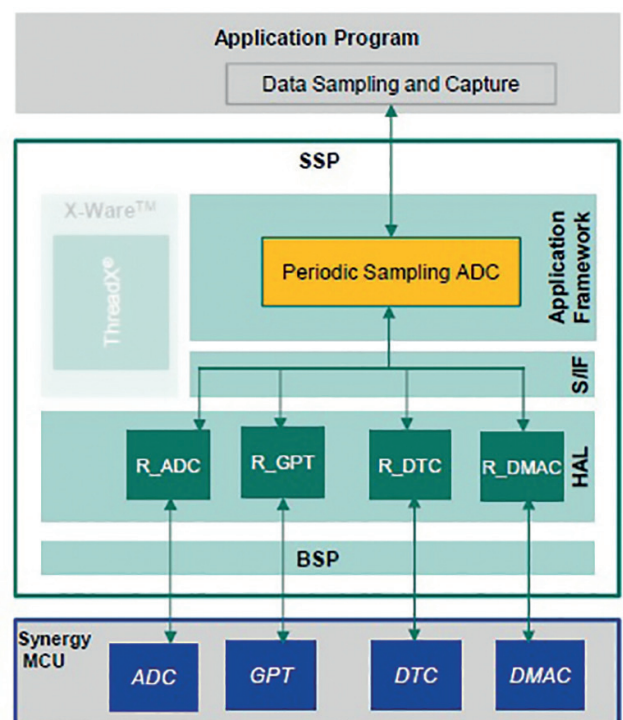


Fig. 75: ADC application framework

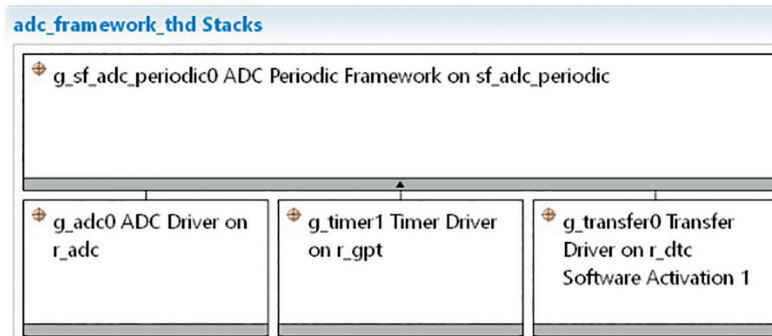


Fig. 76: Stack window with ADC framework

Functional libraries that are included in the SSP are another way of adding high level functions to the SSP for commonly used functionalities. Functional libraries directly interface with the HAL. Like the framework everything is tested and verified – full reliability for rather complex functions and algorithms. And of course full support by Renesas for both the libraries and the frameworks. The functional libraries currently include a DSP as well as a security and cryptography library. More to come...

Did you ever develop your own digital filter or any other more complex signal processing algorithm? And implement it in C code for a microcontroller application? Interesting but painful – in particular if you want to develop an application. But as digital signal processing is frequently used in embedded systems you will for sure need it sooner or later. The CMSIS (ARM Cortex Microcontroller Software Interface Standard) compliant DSP library is automatically added to every project (synergy/ssp/src/bsp/cmsis/DSP_Lib/cm4_gcc/libDSP_Lib.a). It provides many of the most important functions and mathematics to realize efficient digital signal processing algorithms:

- Basic, fast and complex math functions
- Filters
- Matrix functions
- Transforms
- Motor control functions
- Statistical functions
- Interpolation functions

The use of the DSP library is as usual rather simple: add `#define ARM_MATH_CM4` and `include arm_math.h` to your code, use all the functions provided by the DSP library and don't care about the details of the algorithms or the low level code.

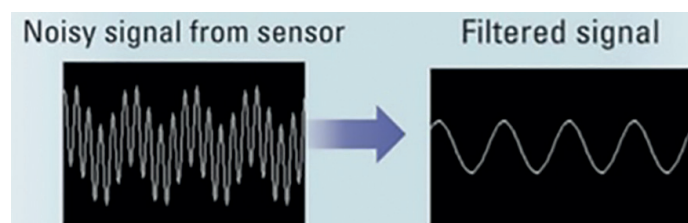


Fig. 77: Filtering of a noisy analog signal using the DSP library

Data security is a major task and concern in any connected application and in particular for IoT applications. The two main security measures for embedded systems are software security and encryption. But encryption for example needs a lot of expert knowledge to program efficient and functional encryption algorithms. You need it, you have to use it, but you still want to focus on your application – so use a library to get the functionality you need.

The crypto interface of the SSP provides many primitive cryptographic operations like listed in Table 7. AES encryption and decryption can be used as an example to show how simple the usage in e²studio is again.

Standard	Description
TRNG (True Random Number Generator)	Generate and read random number
AES (Advanced Encryption Standard)	Encryption and decryption 128-bit, 192-bit, 256-bit key size ECB, CBC, CTR, GCM, XTS chaining modes
RSA (Rivest, Shamir, Adleman asymmetric cryptographic algorithm)	Signature generation, signature verification, public-key encryption, private-key decryption 1024-bit, 2048-bit key size
DSA (Digital Signature Algorithm)	Signature generation, signature verification (1024, 128)-bit, (2048, 224)-bit, (2048, 256)-bit key size
HASH methods	SHA1, SHA224, SHA256

Table 7: Crypto algorithms of S7G2 MCU

AES is a specification for the encryption and decryption of electronic data. With a key length of 192 or 256 bit it is sufficient for top secret information. As it combines a high speed algorithm with low RAM requirements it is very popular for many applications like archive and compression tools, LAN communication and password safe.

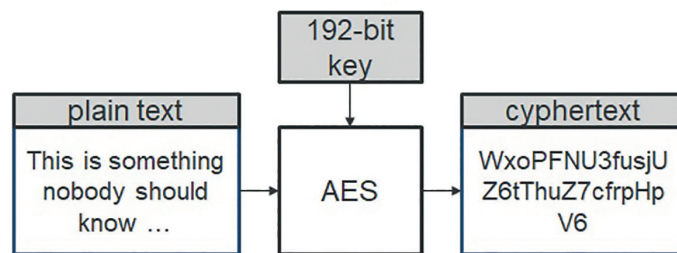


Fig. 78: Schematic of data encryption by a 192-bit AES encryption

In e²studio the crypto modules can be added to a thread by selecting the suitable crypto driver. Properties like key size or chaining mode for the AES driver can be selected in the properties tab. The code in Fig. 79 shows the simplicity to use the API of the crypto driver to encrypt and decrypt data.

```
uint32_t msg_data[32]; /* original data */
uint32_t enc_data[32]; /* encrypted data */
uint32_t dec_data[32]; /* decrypted data */
uint32_t key_data[4]; /* key */
uint32_t ive_data[4]; /* initialization vector data for encryption */
uint32_t ivd_data[4]; /* initialization vector data for decryption */

/* open the secure crypto engine driver */
g_sce.p_api->open(g_sce.p_ctrl, g_sce.p_cfg);

/* open the AES driver */
g_sce_aes_0.p_api->open(g_sce_aes_0.p_ctrl, g_sce_aes_0.p_cfg);

/* encrypt data */
g_sce_aes_0.p_api->encrypt(g_sce_aes_0.p_ctrl, key_data, ive_data, 32, msg_data, enc_data);

/* decrypt data */
g_sce_aes_0.p_api->decrypt(g_sce_aes_0.p_ctrl, key_data, ivd_data, 32, enc_data, dec_data);
```

Fig. 79: Code for encryption and decryption with the AES driver

9. MIDDLEWARE

Application frameworks and functional libraries already provide a high level of abstraction, and in fact both can be regarded to be part of the middleware. A precise definition of middleware is difficult to get, so let's define what it is not: middleware is everything that is not OS, low level drivers or application software. Or, in other words, the line between middleware and application software is blurred and middleware can be regarded as some kind of application software that is taken out of the application software. The reasons for this separation from the application software are manifold.

First of all, like for all predefined modules, commonly used tasks and applications like file handling or GUI driver can be collected in modules of the middleware. The modules provide additional services to the application software and hence a next level of abstraction from the hardware. With regard to flexibility and portability these modules enable a high degree of reusability for different applications and systems. They are ready-to-use and you don't have to reinvent the wheel by writing the same code again and again. As they are developed and tested separately and used in many projects the reliability of the code is high increasing the reliability of the final application. Last but not least the use of middleware provides a higher level of abstraction and simplifies the application code significantly making development of the application faster, simpler and more efficient – focus on application. Taking all these benefits into account the use of middleware can reduce both the development time – enabling faster time to market – and the development cost.

The middleware resides, as the name implies, in the middle of the software, on top of the HAL and BSP and below the application software (Fig. 80). It interacts with the HAL, the BSP, the OS and the application software. You either develop your own middleware (not so often) or you purchase the middleware from a third party vendor. Therefore you have to find a suitable middleware fitting to your requirements and needs. Then you have to check the compatibility to already existing software of your project, like the RTOS or HAL. The third party middleware has to fit to the hardware as well, e.g. with regard to memory requirements and performance.

As any new software stack or module, middleware also introduces some additional overhead. It requires some memory and some performance of the systems – both limited resources in particular for embedded systems. Target for a good middleware is to provide the required functionality with minimum overhead. Scalability of the middleware is also highly appreciated. And the user should just select the functions he needs to minimize the overhead and to keep as many resources for the application as possible.

After you found a fitting middleware, you have to buy it. Here the idea from Renesas is different: within Renesas Synergy the middleware is part of the SSP and hence fulfils all the requirements listed above: it fits to Synergy MCUs and software and is optimized for embedded systems. All middleware components are tested, verified and fully reliable. Even the middleware developed by third party vendors like Express Logic is part of the SSP – free of charge! And for any support for the middleware you just have to contact Renesas, they will take care. So Renesas is the one and only single entry point to the full world of Synergy. As already stated at the beginning of the chapter, even the functional libraries and the application frameworks are in fact middleware, as well as the connectivity modules introduced in the next chapter.

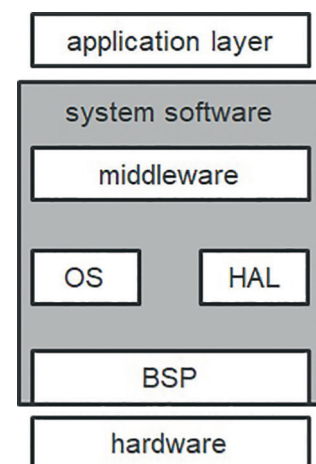


Fig. 80: Location of middleware within system software

9.1. SSP MIDDLEWARE

Like for the RTOS ThreadX® Express Logic is the partner for Renesas Synergy platform for the X-Ware suite of stacks and middleware. Express Logic, located in San Diego, US, is a provider of RTOS and middleware software in particular for IoT applications like consumer devices, medical electronics and industrial control equipment. The software by Express Logic is commonly used worldwide in a large variety of applications, the RTOS ThreadX® has been deployed in over 5.4 billion electronic devices so far!

Within the SSP middleware includes all the components provided by Express Logic:

- FileX® for file-handling
- GUIX™ for design of graphical user interfaces
- USBX™ for USB communication in host and device mode
- NetX™ and NetX Duo™ for network communication

As clearly visible in this list of middleware components they provide rather complex functionalities.

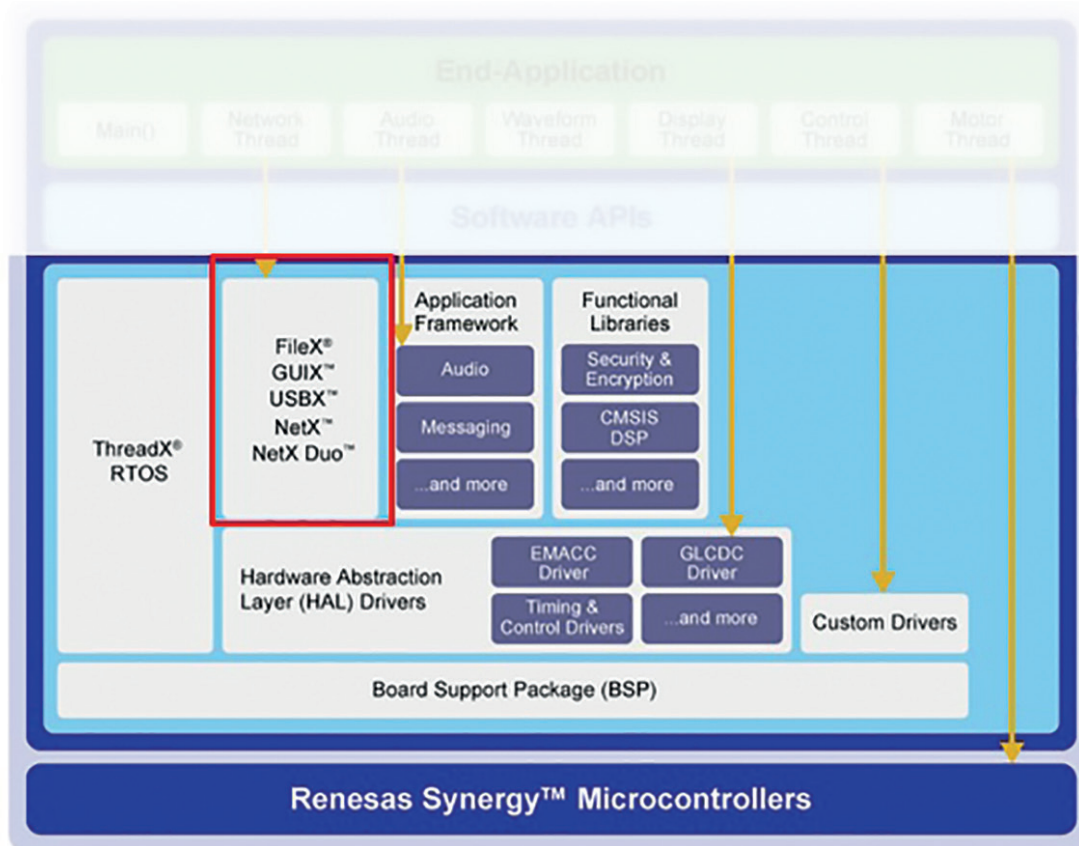


Fig. 81: Block diagram of the SSP with middleware by Express Logic

Besides the benefits given above the middleware components by Express Logic are proven software for commonly used modules. They are optimized with regard to memory footprint and performance. The integration into your system is rather easy, as the modules are directly available within the ISDE – just use them!

FileX® by Express Logic is a MS-DOS compatible embedded file system supporting FAT (File Allocation Table) with 12-, 16- and 32-bits and contiguous file allocation. It also supports an unlimited number of media devices at the same time including RAM disks and flash managers. Each directory entry contains information like attributes (e.g. read-only, hidden), last modified date and time or size of entry in bytes.

FileX® includes many features but only features used by the application are brought into the final software. This approach ensures a high performance and a low memory footprint of about 6 kB. The API of FileX® is easy to understand and easy to use.

Some main features of FileX®:

- Fast execution
- Fast seek logic
- Small footprint
- Internal FAT entry cache
- Contiguous file allocation
- Consecutive sector and cluster read/write
- Long filenames and 8.3 naming conventions
- Unlimited creation of objects like directories and files
- Real-time performance Using FileX®

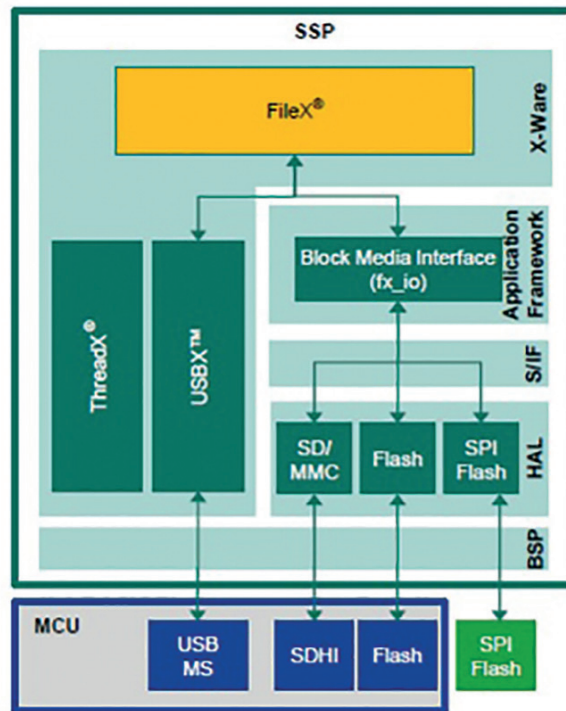


Fig. 82: FileX® embedded file system

You can connect the medium you like to your embedded systems like USB mass storage devices, DS/eMMC cards, flash (SPI, QSPI, On-Chip) or RAM. Fig. 83 depicts the layered structure of a FileX® application providing high flexibility and portability. The Port Block Media Framework (`sf_el_fx`) provides the I/O calls for FileX® and accesses the Synergy Media drivers through the Block Media Framework. The Block Media Framework is the abstraction interface between FileX® and the SSP Block Media. Due to the layered structure and the separation of the application from the media a change of the medium does not change the application code.

Including FileX® to an application is as usual rather simple: just select a thread in the configuration perspective and add X-Ware->FileX->FileX on Block Media for example. Afterwards configure the modules and use the middleware.

Besides connectivity (chapter 10) one of the main features of IoT applications is the interaction between the system and the human operator. In many cases a graphical HMI (Human Machine Interface) is used, like a display or a touch display. Hence more and more GUI features are included in modern embedded applications.

The GUI middleware of choice for the SSP is GUIX® by Express Logic, a high performance GUI framework for real-time graphical applications. This runtime library is implemented as a pure C library and is fully integrated with ThreadX® which allows the design of elegant and functional user interfaces. Like for FileX® it brings only features needed for the application into the final software keeping the memory footprint small. As it is in particular designed for embedded systems it again combines a small memory footprint of about 6 kB with a high performance. The fast and responsive performance is achieved by optimized clipping, drawing and event handling realizing a real-time GUI in the end.

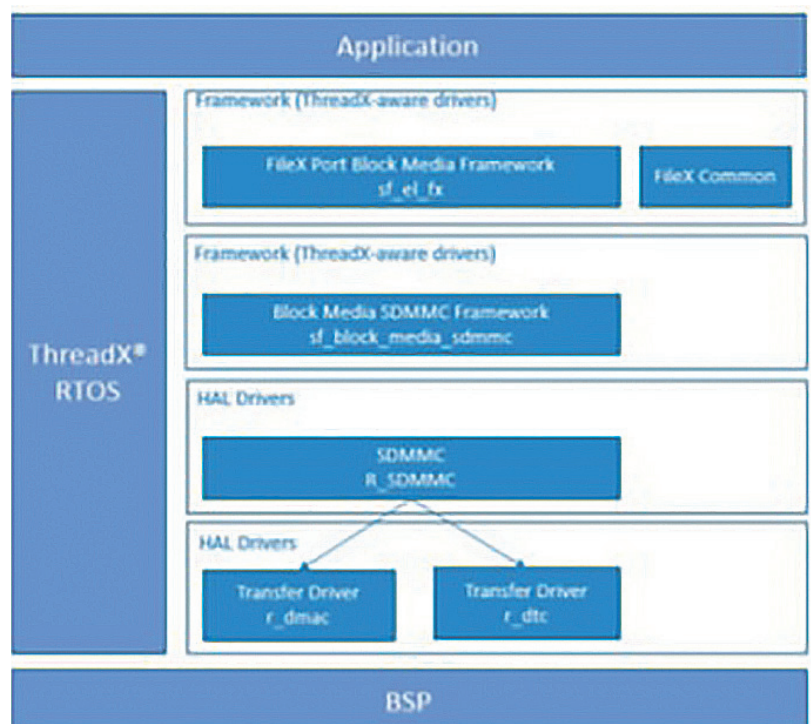


Fig. 83: Block diagram of FileX® application

Features of GUIX™ include:

- High reliability for use in safety critical applications
- Dynamic GUIX™ object creation/deletion of screens, windows, widgets
- Alpha blending and anti-aliasing at higher colour depths
- Multiple canvases and physical displays
- Window blending and fading, screen transitions
- Dynamic animations
- Touchscreen and virtual keyboard support
- Automatic scaling of object size
- Double-buffer toggling control for screen transitions without tearing
- Screen rotations
- Various output colour formats like ARGB888, RGB-888 or RGB565

Fig. 84 depicts the layered structure of the GUIX™ stack. The port module SF_EL_GX in SSP framework layer ties SSP graphics device drivers to GUIX™ by defining an SSP-compliant API and adapting GUIX™ won top of the SSP. The Synergy 2DG engine or the Synergy JPEG engine can be used to accelerate the drawing of widgets.

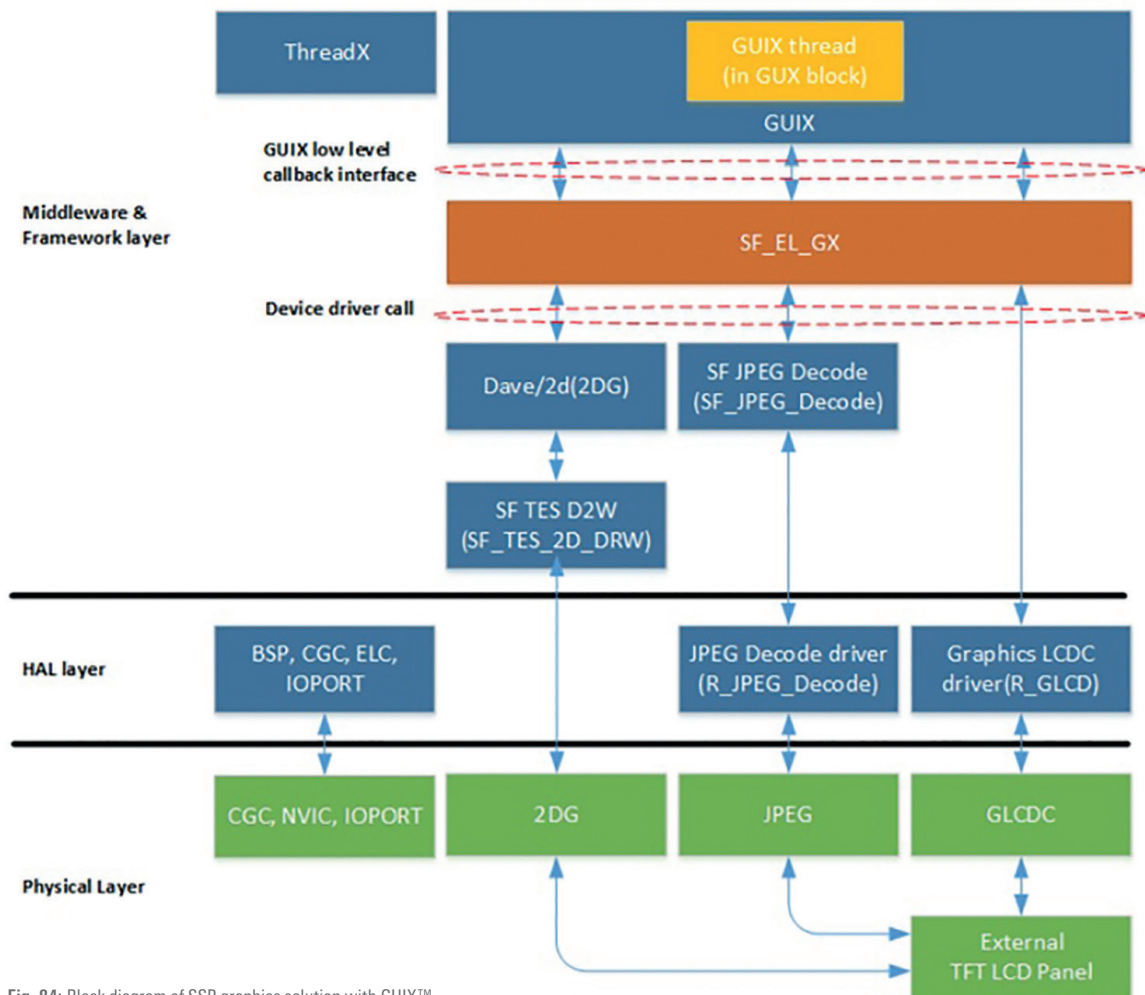


Fig. 84: Block diagram of SSP graphics solution with GUIX™

To design an elegant GUI a design environment is needed as it is rather difficult to design a nice GUI efficiently in pure C code. A dedicated desktop GUI design application, GUIX Studio™, provides a suitable design environment on a standard PC. Graphical user interfaces can be built within the WYSISYG screen easily by drag-and-drop of graphical elements. Graphics from png or jpg files can be imported and converted to compressed GUIX™ pixelmaps. A complete GUIX™ user interface application can be executed on a PC desktop within the GUIX Studio™ environment, allowing a quick and easy generation and demonstration of user interface concepts, testing of screen flows, and observation of screen transitions and animations.

Once the GUI design is finished the target-ready C code for the GUIX™ library can be generated with GUIX Studio™, ready to be compiled and linked with the GUIX and ThreadX libraries. This flow is a fast and easy GUI design and transfer to the target MCU – without any line of C coding.

GUIX Studio™ is a separate program that can be downloaded directly from the Renesas Synergy Gallery (<https://synergygallery.renesas.com/addon/detail/8>). It provides many graphical elements for different application areas like home automation, medical, industrial or consumer applications.

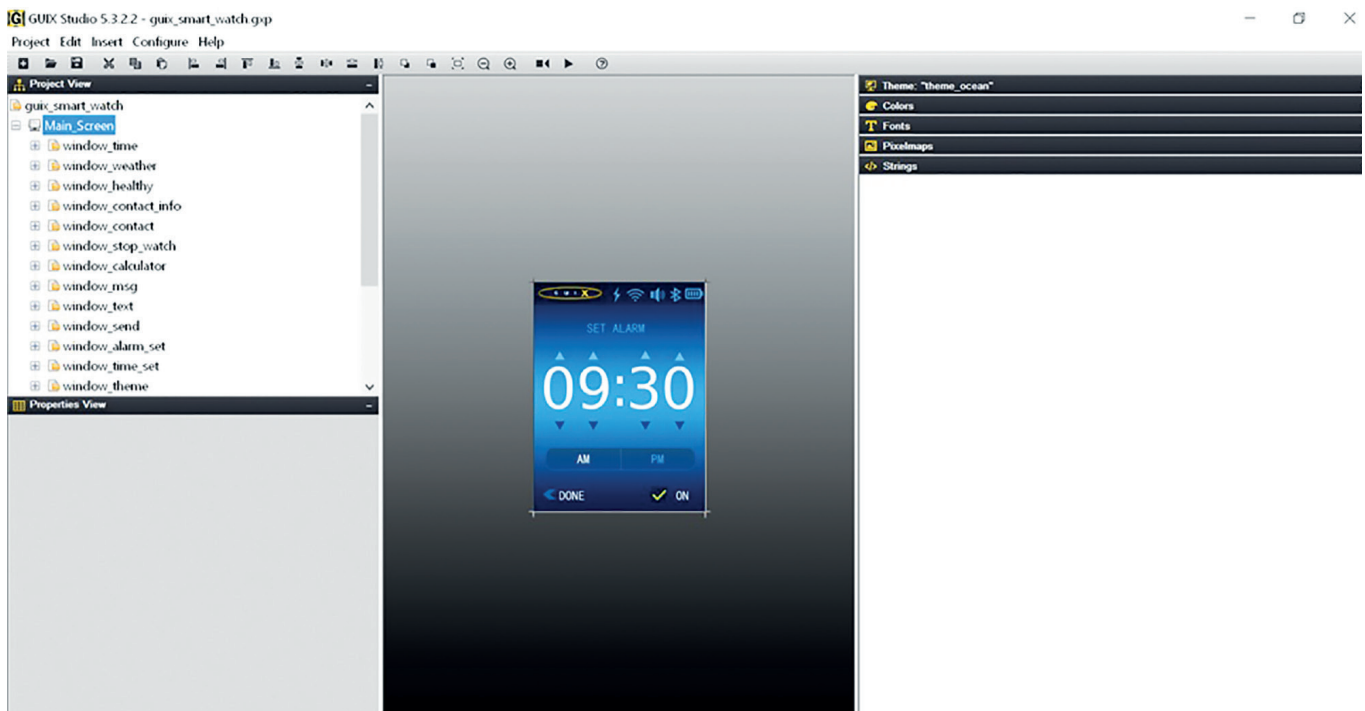


Fig. 85: Screenshot of GUIX Studio™ with a smart watch GUI

Besides FileX® and GUIX™ USBX™ and NetX™ and NetX Duo™ for USB and Ethernet interfaces respectively are part of the SSP middleware. As connectivity is the major topic for IoT applications these modules will be introduced in the next chapter 10.

10. CONNECTIVITY

Connectivity and communication are the key features of any IoT and Industry 4.0 application as everything is connected to everything. But communication is a complex task and manifold and again it is highly appreciated to get best support to set up the communication in these applications. Hence dedicated communication modules or middleware is needed to avoid complex and error-prone programming of the communication stacks.

Let's have a short look at an example for a distributed system with dedicated communication features: a modern car. Besides driving, breaking and steering modern cars provide a bundle of advanced driver assistant systems (ADAS) to make the driving safer, more convenient and more efficient. To realize these ADAS functions many embedded ECUs (Electronic Control Unit) are used inside the car, from motor and gearbox control to emergency braking and lane keeping assist – just a few examples. Even more sensors are integrated into the car to measure everything that is needed for the assistant systems, from simple temperature and pressure sensors to complex radar sensors and cameras. The required assistance functions are not located within one ECU, but they are distributed among several ECUs. Therefore the ECUs and sensors as well as the actuators have to exchange the data needed for realization of the required functionality. As you can imagine – just take the emergency braking as an example – this communication has to be real-time, reliable and failsafe.

Fig. 86 depicts a schematic of a car with several internal networks. Dominant automotive networks are CAN and LIN, but for example Ethernet is getting more and more important. So far cars were mainly internally connected systems, but even cars will be part of the IoT in future as they will incorporate mobile connectivity to the infrastructure and other vehicles. Communication will get more and more important.

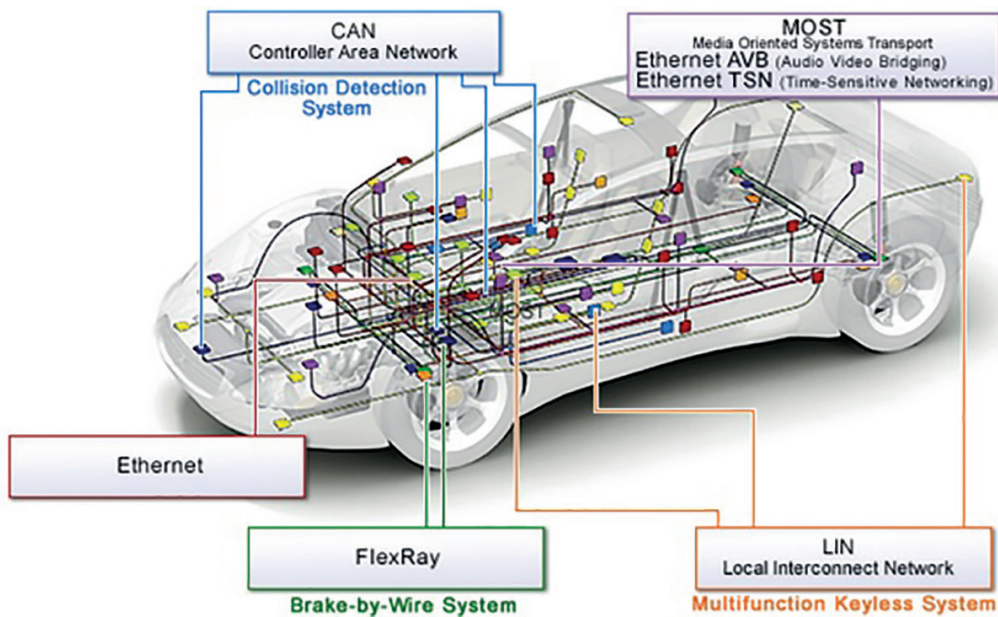


Fig. 86: Data busses of a modern car with distributed systems

The schematic of a distributed system is shown in Fig. 87 taking an ACC system (Adaptive Cruise Control) as an example. This system adjusts the speed of the car automatically according to the traffic situation. It calculates the correct speed based on sensors signals like radar signals, inputs from the driver and data from ECUs like motor and transmission control. Taking all these data into account the ACC ECU brakes and accelerates the car using motor, transmission and braking ECU. A real-time capable and reliable communication between all components is essential to realize the ACC function.

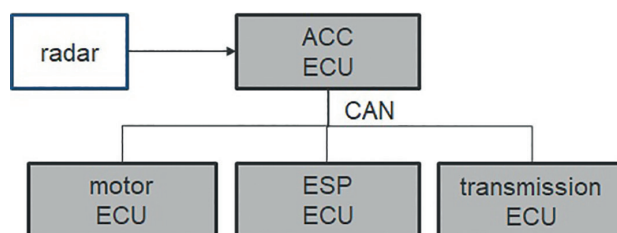


Fig. 87: ACC example of distributed systems

Not only for cars, also for most other embedded systems networking capability is a key feature and a must have. The topic of networking and communication cannot be handled here in detail, but let's have a look at some basic and important characteristics and features.

For data exchange devices are connected via simple point-to-point connections or via bus systems. A bus system is a system for data transmission between several nodes on a common transmission line. Basic characteristics of bus systems are depicted in Fig. 88. There are many different bus systems with different features, characteristics, requirements and transmission media, like CAN, SPI, Ethernet or PROFINET.

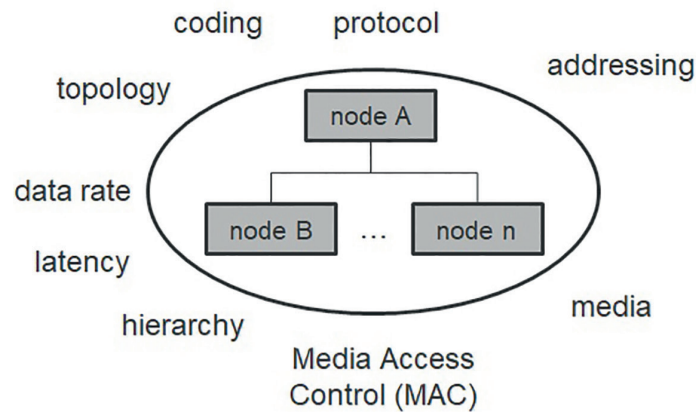


Fig. 88: Basic characteristics of bus systems

Data exchange between different nodes or components can be rather complex as it needs basically the transfer from digital data to a physical representation on the transmission medium and vice versa. Hence a conversion of digital data to a physical representation has to be done. As there are maybe many nodes connected to the bus system and maybe more than just one transmission path, any kind of addressing and routing is needed. Also some kind of protocol is needed to control media access and interoperability. But for the user or application the bus system is of no particular interest, just the user data have to be transmitted – so some kind of abstraction and separation of functionalities is required. Just similar to the layers and abstractions of the software stacks and layers in the modular software.

Imagine the networking you are using every day, the internet. You want to use your applications like mail, web browser or anything else. You don't want to care about how the data are transmitted, cable, optical fibre or wireless LAN. It just should work and meet your requirements with regard to performance and data rate.

To achieve this kind of abstraction the famous OSI model (Open Systems Interconnection) is used. This model standardizes communication functions without regards to underlying technology or media (IOS/IEC 7498-1) to ensure interoperability of diverse communication systems and to define standard protocols.

In the OSI model the communication system is partitioned into seven abstraction layers (Fig. 89). A layer serves the layer above it and is served by the layer below it. Each of the seven layers has a specific sub-function of the communication, e.g. conversion of digital to analog signals in layer 1, the physical layer. Between the layers there are standardized interfaces. Remember the modular software stacks with standardized interfaces, rather similar, isn't it?

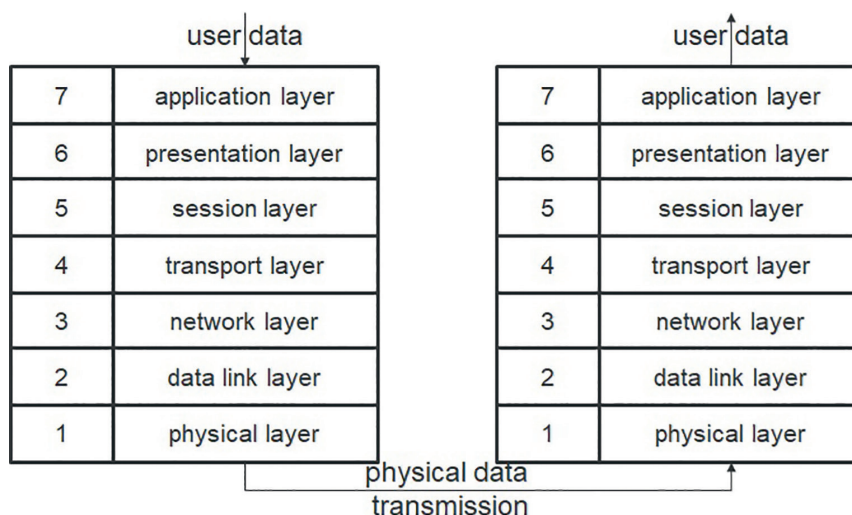


Fig. 89: OSI model of data communication

From digital user data in the first node to digital user data on the second node the data pass the layers top down in the sending node and bottom up in the receiving node. When receiving data from the higher layer each layer of the sending node adds some control data to the received data. This overhead contains important information for the corresponding layer on the receiving mode but doesn't care for the lower layers, refer to Fig. 90. Layer 4 receives data from layer 5 called SDU 4 (Service Data Unit). Layer 4 adds its overhead OH4 to build the PDU 4 (Protocol Data Unit). Afterwards this PDU 4 is forwarded to layer 3 and layer 3 interprets these data as SDU 3 – the overhead OH4 is not important for layer 3. So the PDU of layer n+1 gets the SDU of layer n in general.

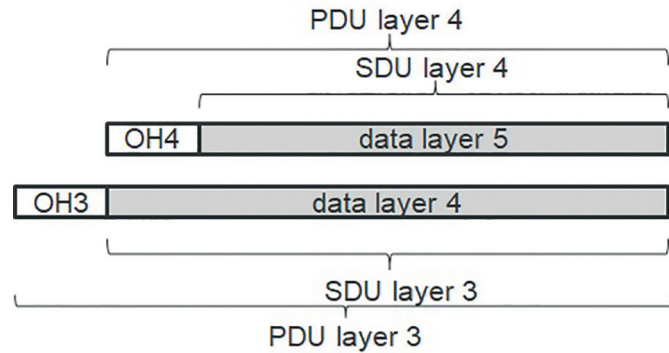


Fig. 90: PDU and SDU including the overhead

The function of each layer is clearly defined in the OSI model (Fig. 91 and Fig. 92). The example of the TCP/IP protocol in Fig. 92 clearly demonstrates the separation of functionalities and some well-known buzzwords.

host layers	7	application layer	high-level API
	6	presentation layer	translation of data between networking service and application
	5	session layer	management of communication sessions
	4	transport layer	reliable transmission of data segments
media layers	3	network layer	management of multi-node network, e.g. addressing, routing
	2	data link layer	transmission of data frames between two nodes
	1	physical layer	transmission and reception of raw bit streams on physical medium

Fig. 91: Functions of the OSI model layers

Ethernet is the dominant connectivity system worldwide and defines the layers 1 and 2 only. What makes it a success story are the layers on top of these two Ethernet layers, famous TCP/IP layers (3 and 4) and the application related layers providing functions like HTTP (Hypertext Transfer Protocol) or SMTP (Simple Mail Transfer Protocol).

host layers	7	application layer	FTP/HTTP/SMTP/Telnet/...
	6	presentation layer	
	5	session layer	
	4	transport layer	TCP/UDP
media layers	3	network layer	IP
	2	data link layer	Ethernet MAC
	1	physical layer	Ethernet PHY

Fig. 92: OSI model of TCP/IP internet protocol

Whether all layers are implemented and used in a bus system depends on the bus system, not all layers have to be used. Internet uses all seven layers as shown above whereas CAN for example just implements layers 1 and 2. For CAN the physical layer transforms the digital signals into differential signals and vice versa. This layer is realized in a separate and dedicated hardware, the CAN transceiver (Fig. 93). The data link layer (layer 2) is often realized as a dedicated hardware module of a microcontroller, like for most CAN systems. The interface between the two layers is standardized and for CAN this interface is built from just two signal lines (TX and RX) for the transmission of receive and transmit data. As the two layers are implemented in different components the standardized interface ensures the combination of different devices of different vendors.

Layers 3 to 7 are not implemented for pure CAN communication (there are some layer 7 implementations for CAN like CANopen for automation systems). In general layers 3 to 7 are pure software layers and the set-up of the communication requires a lot of knowledge of the bus system and takes a lot of effort. Therefore the use of middleware or low level communication modules to simplify the set-up of the communication and the communication itself is highly appreciated.

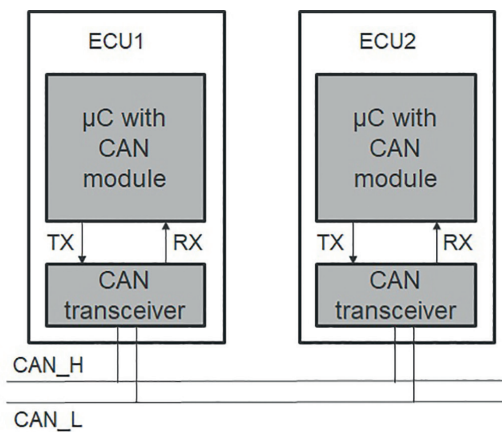


Fig. 93: Part of CAN bus with two nodes and differential transmission line

10.1. SSP CONNECTIVITY

The Renesas Synergy™ Platform provides a manifold bundle of connectivity solutions and bus systems that provide the required interoperability between the hardware and software for complex communication systems. The hardware has dedicated hardware modules that implement communication features like the data link layer of CAN communication. In addition the hardware provides the required port functions for the communication with the external components. These port functions can also be used for bus systems like I²C to realize layer 1 functions.

The software modules for communication can be found all over the SSP (Fig. 94). The HAL provides low level drivers like the CAN data link layer or the Ethernet MAC controller. Sophisticated communication solutions are available within the application framework, e.g. for SPI or I²C communication. Most complex connectivity systems are realized within the middleware using modules from Express Logic like NetX™.

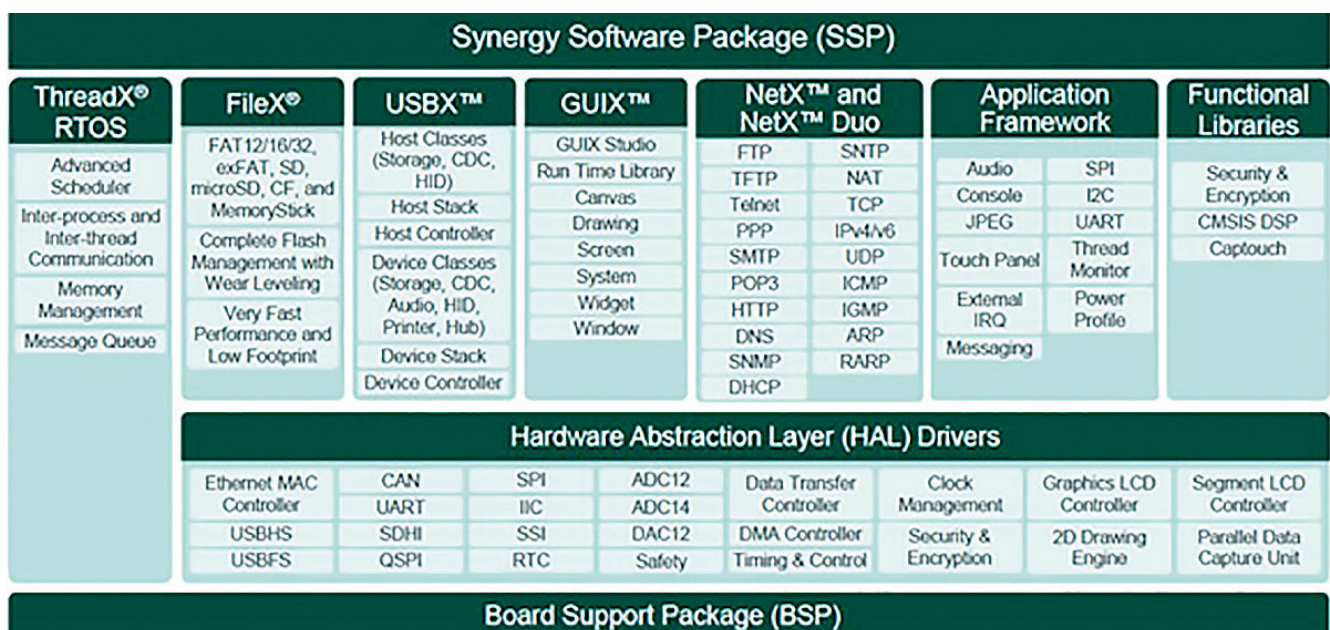


Fig. 94: SSP with communication modules

In the next sections some of the bus systems will be introduced, I²C, Ethernet and USB, starting with a short general introduction for each bus system followed by the Synergy implementation.

10.1.1. I²C

I²C (Inter-Integrated Circuit, IIC) is a serial and synchronous multi master bus system. It is mainly used for communication between ICs and/or sensors with data rates up to 5 Mbps. Just layers 1 and 2 are specified for this commonly used bus system. In layer 1 data transmission is done on 2 wires, a clock (SCL) and a data line (SDA), using non-return-to-zero coding (NRZ). The wires are directly driven by a microcontroller pin in open-drain configuration. The bus topology is defined in layer 2 as well as the communication protocol (Fig. 95). ICs use direct addressing of the bus nodes with a 7- or 10-bit address and data transfer is done byte by byte.

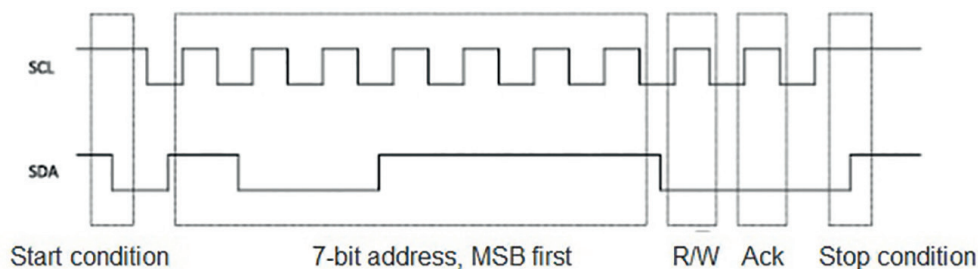


Fig. 95: I²C communication with SCL and SDA line

I²C implementations can be found at different parts of the SSP.

First the HAL provides I²C modules. Two RIIC drivers implement two separate HAL interfaces for I²C, one implements the I²C master interface (RIIC HAL) and the other one implements the I²C slave interface (RIIC Slave HAL). Both RIIC drivers are interrupt driven and support I²C normal-mode (up to 100 kbps), fast-mode (up-to 400 kbps) and fast-mode plus (1 Mbps, on selected channels of S7G2 and S5D9) and 7- and 10-bit addressing mode.

Also the SCI peripheral of the MCU can be used to implement an I²C communication. Using this SCI_I2C driver I²C communication can be done in master mode only with a transfer rate up to 400 kbps (fast-mode) and 7- or 10-bit addressing. Memory requirements for each HAL I²C module is about 5 kB. These modules are available just by adding a new stack Driver->Connectivity to a thread. Configuration is done in properties tab as usual.

I²C is also part of the application framework. The I²C framework is Thread™ aware and abstracts the software interface for the I²C driver. It handles the integration and synchronization of multiple I²C peripherals on the I²C bus. It uses the low-level I²C driver modules and SCI common driver modules for I²C communication.

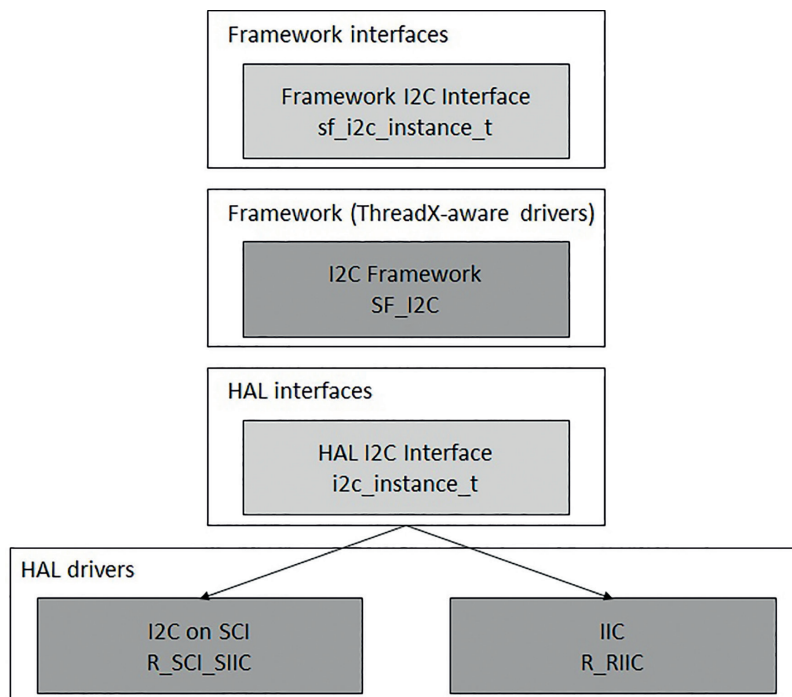


Fig. 96: I²C framework stack

10.1.2. ETHERNET

Ethernet is the dominant bus system for all kind of data and PC networks all over the world. The Ethernet standard IEEE 802.3 just specifies layers 1 and 2 of the OSI model. Several sub-standards detail dedicated Ethernet types like 802.3u for Fast Ethernet or 802.3z for Gigabit Ethernet over optical fibre. There are many success factors for Ethernet. One of the key success factors for Ethernet is the strict separation of layer 1 and 2. As depicted in Fig. 97 the data rates increased drastically in the last years reaching 100 Gbps. Another success factor is the flexibility with regard to the transmission media. Totally different media can be used for different Ethernet specifications, like optical fibre or twisted pair cable. Layer 2 is, due to the strict separation from layer 1, independent of the media used and of the data rate. A third success factor are the additional layers specified to provide a high level of abstraction and services, like TCP/IP or HTTP.

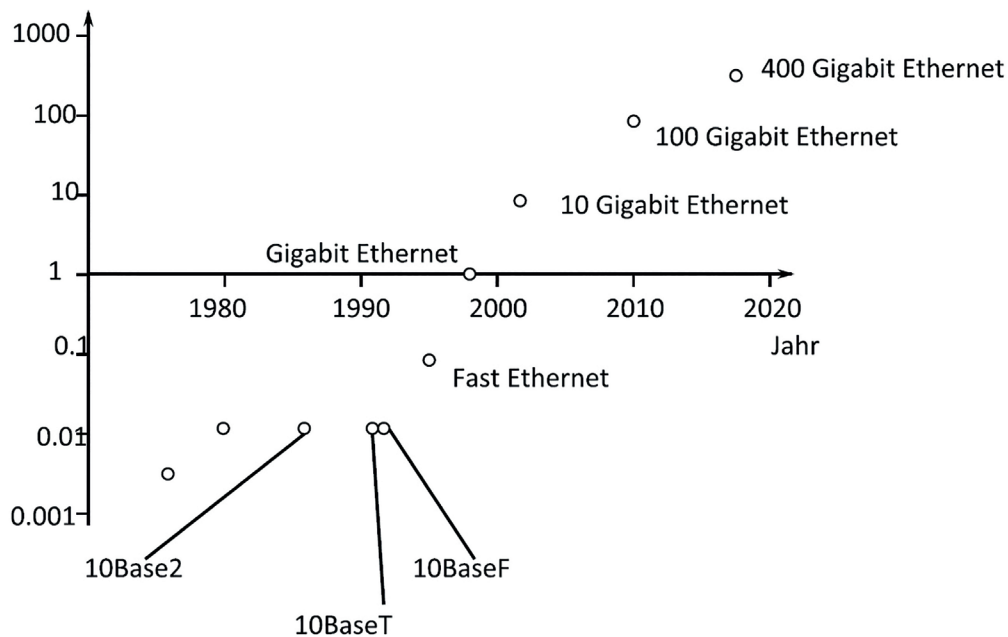


Fig. 97: Evolution of data rate of Ethernet

Some basic features of Ethernet:

- Node addressing with fixed address per node
- Event driven communication
- Multi master
- Several hierarchies
- Media include coax cable, twisted pair cable, radio, fiber optics
- Media access according to CSMA/CD (Carrier Sense Multiple Access/Collision Detection)

Due to the event driven CSMA/CD there are always collisions of messages possible when two or more nodes start transmission of data at the same time. As the collisions lead to a stop of transmission followed by a restart of transmission some time later there is an undefined latency for data transmission. Hence Ethernet communication is non-deterministic and does not provide real-time capability. These limitations have to be considered when setting up Ethernet communication in real-time applications. There are some mechanisms to make Ethernet more or less real-time capable, in particular for automotive applications real-time enhancements like BroadR-Reach are under development.

For addressing in an Ethernet network each node gets a unique 48-bit address called MAC address (Fig. 98). MAC addresses are allocated by an IEEE organisation. In local networks it is also possible to allocate individual MAC addresses that are unique only within the local network.

I/G	U/L	Organisationally Unique ID	Device ID
1 bit	1 bit	22 bit	24 bit

Fig. 98: Structure of a MAC address

An Ethernet frame is 64 to 1519 bytes long and contains a maximum of 1500 bytes of user data and control data (Fig. 99). For unique identification of sender and receiver of a frame it contains the MAC source and MAC destination address. A CRC field (Cyclic Redundancy Check) is used at the end of the frame to protect against common types of errors during transmission and to assure the integrity of the frame with a Hamming distance of 3.

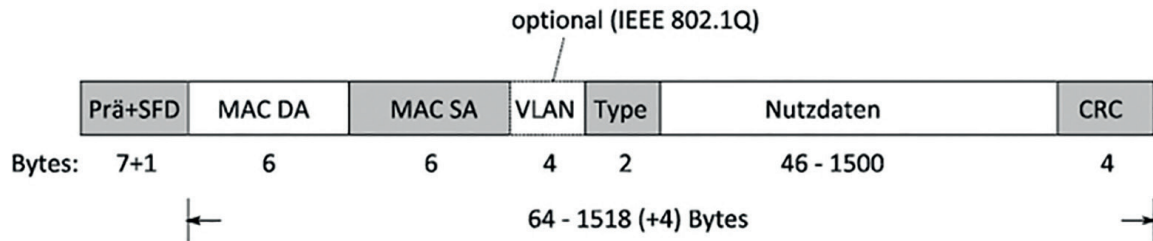


Fig. 99: Structure of an Ethernet frame

Like for CAN the hardware implementation of Ethernet reflects the strict separation of layer 1 and 2 (Fig. 100). The Ethernet MAC (layer 2) is part of a microcontroller like the S7G2. The physical layer is implemented in a separate PHY. The interface between the two components is standardized, e.g. in terms of a MII interface (Media Independent Interface) for Fast Ethernet (100 Mbps) in IEEE 802.3u. Due to this separation different types of PHY devices for connecting to different media like twisted pair cable or optical fibre can be used without changing the MAC hardware.

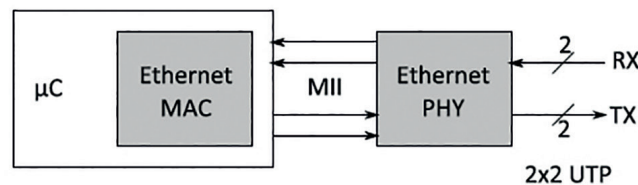


Fig. 100: Separation of Ethernet MAC and PHY

The protocols on top of the Ethernet layers define the function you want to use and are the basis for many applications (refer to Fig. 92). The IP layer 3 (Internet Protocol) defines the routing of the data through the network in packets. The routing and addressing is done based on IP-addresses. Each device has a unique logical IP-address, e.g. 160.54.16.132 for IPv4. Currently two IP standards are in use, IPv4 (RFC 791) and IPv6 (RFC 2460). The new IPv6 standard uses 32 characters for the address (IPv4: maximum 12) resulting in an incredible number of unique addresses – $340 \cdot 10^{36}$ addresses – more than enough to connect everything you can imagine to the internet. Welcome to the Internet of Things!

For layer 4 of the OSI model two different protocols are standardized and used. TCP (Transmission Control Protocol, RFC 1323 standard) performs a host-to-host communication and provides the channels for the communication needs of an application. To ensure data integrity during transmission the data are checked for errors and the successful transmission is controlled. As the latency is rather high the use in real-time applications is difficult – at least if the transmitted data are needed in real-time. Together with the IP layer it forms the famous TCP/IP internet protocol stack and is used for all kind of internet applications like www and mail.

The second protocol of layer 4 is UDP (User Datagram Protocol, RFC 768). In contrast to TCP there is no error checking and no reliability with regard to successful data transmission. The latency of data transmission is lower than for TCP. Main application field for UDP are multimedia applications.

On top of the TCP and UDP layer are the application layers that provide user functions for distributed systems:

- HTTP (Hypertext Transfer Protocol): web server and network management
- POP3 (Post Office Protocol), SMTP (Simple Mail Transfer Protocol): e-mail transfer
- FTP (File Transfer Protocol), Telnet: connectivity and data transfer protocol

Obviously Ethernet and the higher layers are complex topics and the proper setup is rather difficult. Therefore a dedicated middleware to provide the required functionality is highly appreciated. Express Logic's NetX™ and NetX Duo™ implement a TCP/IP protocol stack for the SSP to enable IoT and M2M communication protocols. This middleware is optimized for embedded systems with regard to performance and memory requirements as only services needed in the application are included in the final software (Fig. 102). NetX™ and NetX Duo™ are completely integrated with ThreadX® and with TraceX® real-time analysis support.

NetX™ supports IPv4 and provides an BSD (Berkeley Software Distribution) socket compatible API. It provides a set of protocol components that comprise the TCP/IP standard, e.g. TCP, UDP and ARP (Address Resolution Protocol, RFC 826). Additional applications of layer 5 to 7 are available on top of NetX™. This application bundle includes the protocols listed above (HTTP and Co) as well as DHCP (Dynamic Host Configuration Protocol) and DNS (Domain Name Server) and many more. In fact NetX™ is a subset of NetX Duo™.

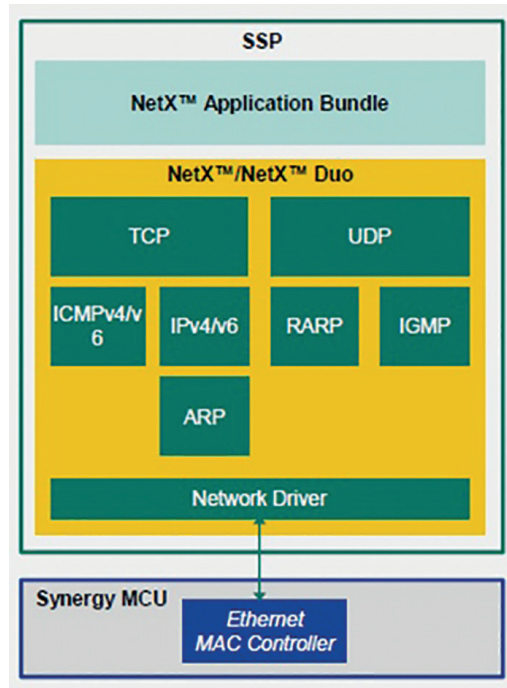


Fig. 101: NetX™ and NetX Duo™ middleware

NetX Duo™ includes all features of NetX™, in addition it supports both IPv4 and IPv6 and additional features like IPsec with IKEv2 (Internet-Key-Exchange protocol) for secure communication and duplicate address detection. NetX Duo™ is accredited by the IPv6 forum with Phase-II IPv6-Ready Logo certification. It is also certified by SGS-TÜV Saar for use in safety-critical systems including medical devices, process control systems, industry machinery and automotive applications (Table 8). The use of a verified and certified software provides many benefits for the user and the application again: reliability, efficiency, cost reduction, shorter time-to-market.

Module	ROM Size (bytes)	RAM Size (bytes)
IP	3996	140
Packet	1040	8
ARP	2050	0
UDP	2220	0
Total	9306	148

Fig. 102: Memory usage of an IoT application using UDP

Certificate	Description
IEC-61508 SIL 4	Functional safety of electrical, electronic and programmable electronic safety-related devices and systems
IEC-62304 SW Safety Class C	Medical device software
ISO 26262 ASIL D	Road vehicles functional safety
EN 50128	Safety relevant software for railway

Table 8: Certificates of NetX Duo™

10.1.3. USB

USB (Universal Serial Bus) is a bus system for communication and power supply between computers and electronic devices using standardized connections including cables, connectors and protocol (IEC 62680). The USB is simple to use (hot pluggable just like your flash memory stick), reliable and provides a high data rate. In addition it is possible to power devices over USB – just like the connection from your PC to the Starter Kit. This bus is commonly used worldwide for all kind of electronic devices such as PC, peripherals (keyboards, printers, disk drives) and mobile devices (digital cameras, smartphones, PDA).

The design architecture of USB is an asymmetrical master-slave architecture (Fig. 103). The host, e.g. a PC USB, acts as a master for the USB communication and it controls the complete communication. On the other hand the peripheral devices act as slave of the USB communication. They just act after request of the USB master. As is well-known from your PC there is always one device per USB port, e.g. a keyboard or memory stick.

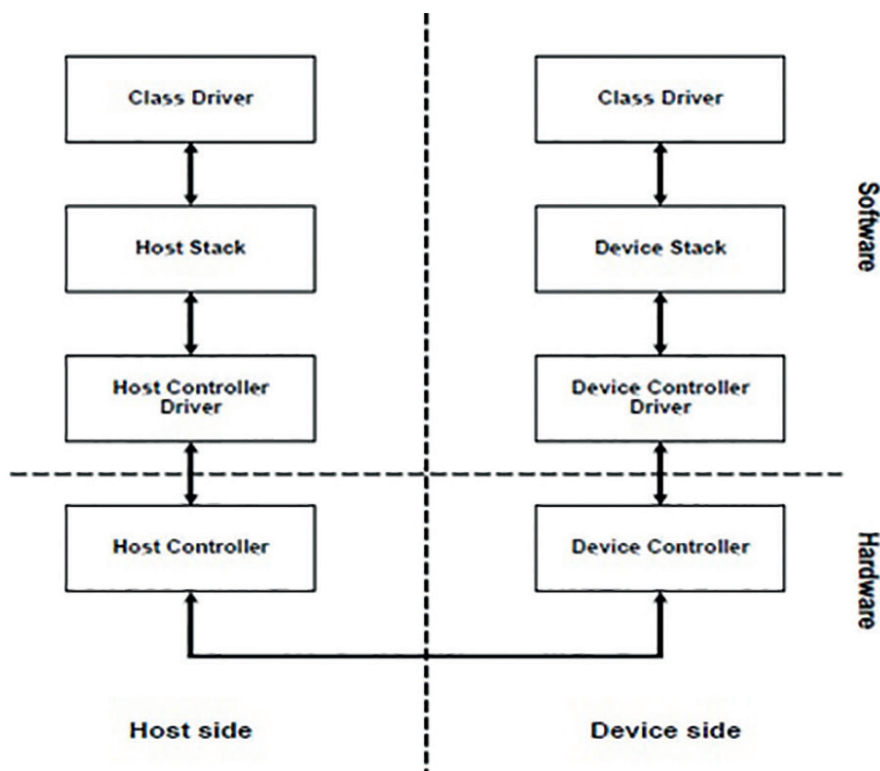


Fig. 103: Architecture of USB

Currently two USB specifications are used, USB 2.0 and USB 3.0. USB 2.0 supports data rates up to 480 Mbps in high speed mode. The maximum cable length at high speed is about 5 m. Devices connected to USB 2.0 can be supplied with up to 500 mA. To ensure compatibility with older devices it is backward compatible to USB 1.1. USB 3.0 provides significantly higher data rates of up to 5 Gbps for cable length of up to 3 m. Up to 900 mA power supply for connected devices is supported. Again it is backward compatible, this time with USB 2.0.

There is a great manifold of devices that can be connected to a USB host. To avoid the need for unique drivers for each device, devices are clustered in classes. Each class can be driven by a generic driver, no specific driver is needed. These classes include HID (Human Interface Device) like keyboard or mouse, CDC (Communications Device Class) like Wi-Fi adapter or modem and Storage devices like USB stick, harddisk or MP3 player.

The SSP uses USBX™ middleware from Express Logic to enable high level USB functionality. It is again optimized for small memory footprint and high performance and is completely integrated with ThreadX®. It provides both a host and a device stack. USBX™ supports USB 1.1 and USB 2.0 specification with a maximum data rate of 480 Mbps.

The host mode is used if the Synergy MCU acts as a USB master. The USB core stack detects the insertion and removal of devices and is responsible for protocols. The USB controller supports major USB standards like OHCI (Open Host Controller Interface) and EHCI (Enhanced Host Controller Interface). Supported USB standard classes include HID, CDC, storage and printer.

The USB device stack is used for the USB slave mode of the Synergy MCU. Again standard device classes are supported (CDC, HID, storage, ...).

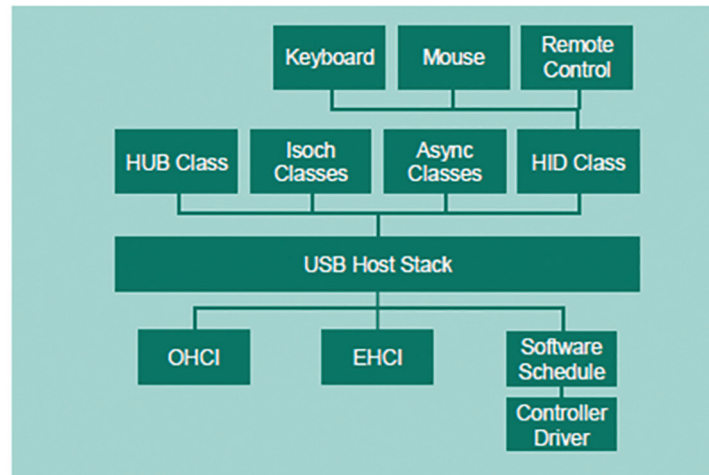


Fig. 104: Block diagram of USB host mode stack

As we have seen the SSP provides many communication modules to connect the Synergy MCU to other devices. Hence setting up a communication is getting rather simple – select the required driver, framework or X-Ware in the corresponding thread, add it to the thread and configure it in the properties view. For your application code just use the API function calls to get the required functionality.

And just in case you want to secure your communication to prevent any unauthorized access to your communication data or device – just use the security and encryption framework (chapter 8). This framework supports your need for secure data transmission.

11. RENESAS SYNERGY PLATFORM

Finally we have everything to realize embedded applications – from hardware via development tools to sophisticated software stacks and frameworks. But where to get all this stuff?

The traditional way of getting everything you need is a rather long way: decide what you need for your application, buy hardware with some basic software from a semiconductor company or distributor, check availability and compatibility of components like IDE and software stacks, purchase these software components, bring everything together and check whether it works in general, and finally, after all these steps – start to develop your application. So many steps, companies, negotiations and communication – contact the corresponding supplier separately. By the way, what about the reliance on all these suppliers and the different life cycles of the components. All these topics are difficult to handle. And it's getting worse if you need support for anything – and support from the supplier is one of the key requirements of design teams to get all the help they need.

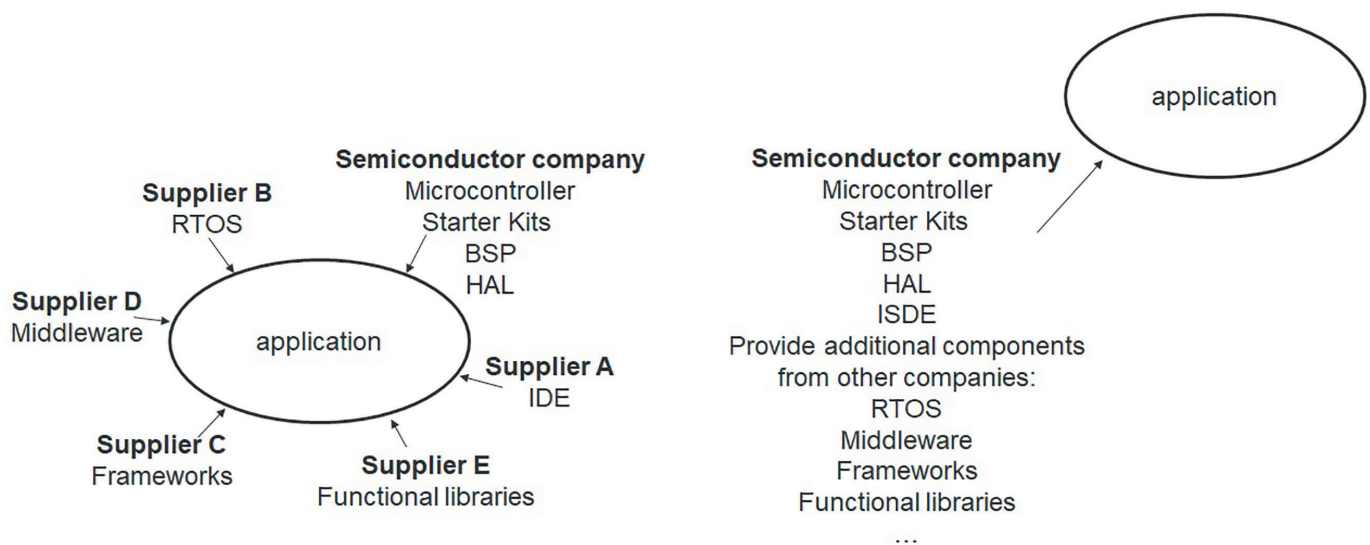


Fig. 105: Traditional schematic of suppliers for an embedded development (left) and Renesas Synergy™ Platform (right); each arrow indicates separate negotiations, support, communication, ...

Taking all these topics into account is the basic idea of the Renesas Synergy™ Platform. This platform is a one shop and one stop solution. With just one shop you have just one reliable supplier for (nearly) everything that provides harmonized components, software, tools and solutions. One stop means you have a single entry point for everything you need, just one communication channel and one negotiation partner – a great simplification of all the annoying topics you have to deal with before starting any development. Target of this solution is: focus on application!

- Simplify development and entry
- Speed up development
- Faster time to market
- Reduce overhead
- Reduce cost
- Provide the ground work and relieve the customer

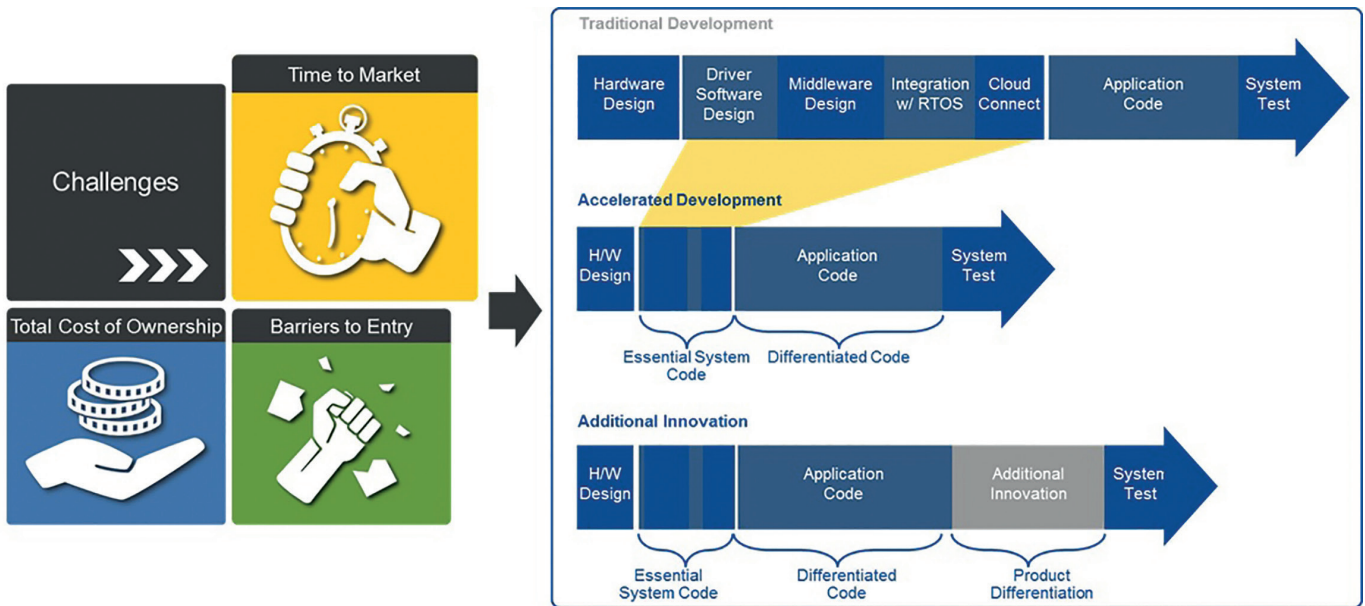


Fig. 106: Faster time to market and focus on application with Renesas Synergy™ Platform

The Renesas Synergy™ Platform is a qualified platform (Fig. 107): best practices are taken into account, the software data sheet is the basis of the SSP warranty, it follows well-respected industry standards for software development and follows a strict software quality assurance plan.

Best Practices	Software Data Sheet	Industry Standards	Software Quality Assurance (SQA)
<p>Software Development Life Cycle (SDLC):</p> <ul style="list-style-type: none"> ▪ Renesas SDLC guideline document ▪ Requirements & Traceability ▪ Coding Standards ▪ Design Descriptions ▪ Code Reviews and Unit Test Development ▪ Continuous Integration and Integration Reports ▪ Release Process & Management 	<p>For Synergy Software Package (SSP) on multiple hardware platforms:</p> <ul style="list-style-type: none"> ▪ Published and maintained on Renesas.com website ▪ Specs and performance metrics tested and documented ▪ Benchmarks, code size, context switch times, latencies, execution times, cyclical testing, fault tolerance and more. ▪ Basis of SSP warranty 	<p>Well-respected standards for software development:</p> <ul style="list-style-type: none"> ▪ MISRA C:2012 – Guidelines for the Use of the C Language in Critical Systems ▪ ISO/IEC/IEEE 12207 – Software life cycle processes ▪ Testing artifacts available for process certification - TUV, UL 	<p>Professional software:</p> <ul style="list-style-type: none"> ▪ Renesas SQA document – Software Quality Assurance Plan ▪ Requirements traceability throughout development ▪ Documented processes ▪ SQA metrics & process artifacts available to customers ▪ Test plans, test suites, reports, software quality handbook

Fig. 107: Renesas Synergy: a qualified platform

The Renesas Synergy™ Platform consists of five main elements like depicted in Fig. 108. The software was already introduced in previous chapters. The Synergy Software Package (SSP) including BSP, RTOS, HAL, middleware and libraries is maintained and supported by Renesas, even the third party software like the Express Logic middleware. The SSP is warranted by Renesas to operate within the specifications of the published datasheet. Each release of the SSP is qualified according to a Software Quality Assurance process developed based on ISO/IEC/IEEE 12207 standards.

In addition to the SSP software Renesas additional software in terms of Qualified and Verified Software Add-Ons (QSA, VSA).

The Qualified Software Add-Ons augment the SSP with additional software functionality like specialized connectivity stacks of control algorithms. These software components are fully SSP compatible and are developed and tested under the same rigorous quality requirements of the SSP. They are sold, licensed and services by Renesas – remember the one shop, one stop solution.

The Verified Software Add-Ons are third party products from best-in-class industry experts also extending the functionality of the SSP. The compatibility with the SSP is proven and the software has been tested on Renesas hardware. VSA evaluation software can be downloaded from the Renesas Synergy Gallery (see below). The final VSA software is licensed from the software vendor (whether free or chargeable depends on the software vendor), but still Renesas supports the initial assessment and ensures that the best possible support is given.






Software	Microcontrollers	Tools & Kits	Solutions	Gallery
<ul style="list-style-type: none"> ▪ Qualified Synergy Software Package (SSP) for guaranteed operation ▪ Complete package fully integrated and maintained ▪ Applications can be written at the software API level 	<ul style="list-style-type: none"> ▪ Wide MCU spectrum based on 32-bit ARM® Cortex®-M CPU cores ▪ Completely scalable and pin compatible ▪ On-chip Flash memory up to 4 MB ▪ Security & encryption acceleration ▪ Ultra low power 	<ul style="list-style-type: none"> ▪ Integrated Solution Development Environment (ISDE) with context-aware documentation ▪ Starter Kits (SK) and Development Kits (DK) for immediate demonstration of entire software package 	<ul style="list-style-type: none"> ▪ Product Example (PE) kits: Complete design journeys representative of end-product designs ▪ Application Example (AE) kits: Technology building-block examples 	<ul style="list-style-type: none"> ▪ Web access to Synergy specific software, tools, licensing, plus 3rd party software & services ▪ Future growth to complete, secure cloud access infrastructure 

Fig. 108: Main elements of Renesas Synergy™ Platform

The MCU is of course the basis for any application, and the Synergy MCUs are in particular dedicated to the Renesas Synergy Platform. These 32-bit MCUs build a complete family of four series of devices and combine high performance with low power and provide a high degree of scalability and flexibility – just select the hardware you need.

The set of development tools, kits and design examples includes everything you need to start an efficient and reliable development until your final code. Besides the ISDE e²studio also the IAR Embedded Workbench can be used as development environment. In addition to the Starter Kits also development kits are available. These kits are complete development platforms to explore all capabilities of the microcontroller. They are also the basis for the qualification of the SSP.

Even more sophisticated kits are available within the Synergy Solutions part. Here actual product examples (PE) and application examples (AE) bundle different technologies to provide solutions for more sophisticated applications. A product example is an implementation of a particular end product. It can be reused for customer products – learn from the experts and speed up your design! These product examples include the Synergy MCU, SSP, QSA and VSA as well as schematics, a bill of material and the design documentation – everything you need as a starting point for your own application and system.

The PE-HMI1 product example for instance offers a unique perspective into the design of a connected HMI, closely representing how an end-product would be designed using the Synergy Platform. This particular PE features a brilliant 7-inch WVGA colour TFT display with capacitive touch screen and a CMOS imager that makes use of the graphics capability of the Synergy S7G2 MCU. Included are the basic software components for the whole world of connectivity – Ethernet, USB, WiFi, Bluetooth® and BLE® as well as security. It can serve as a starting point for access control, home automation or other connected HMI applications for example.

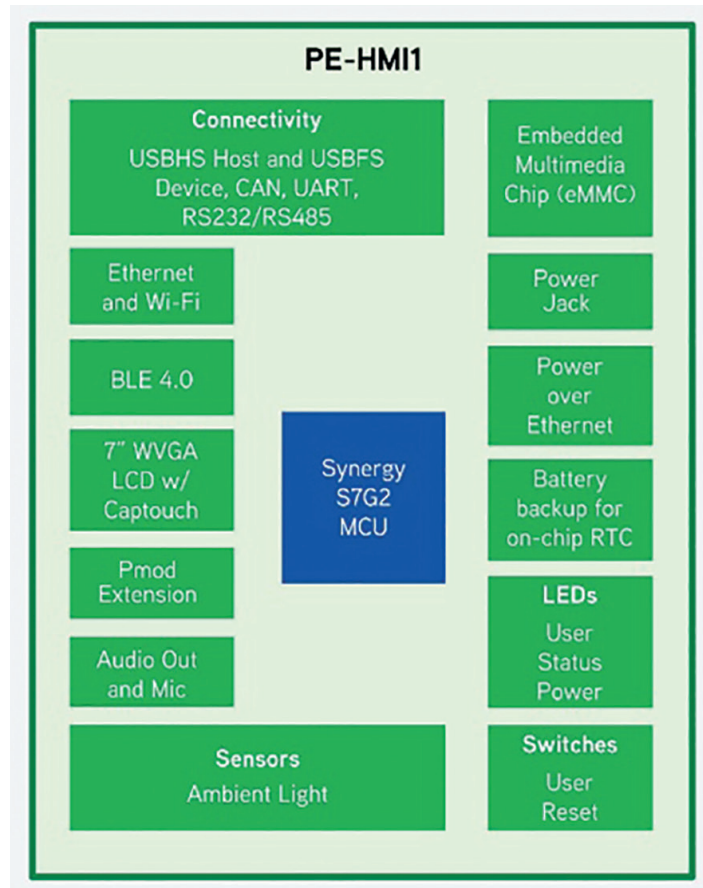


Fig. 109: PE-HMI1 block diagram

AE-CAP1 is an application example for a capacitive touch application. This kit includes five PCBs, two CPU boards (S3A7 MCU and S125 MCU) and three sensor boards. Primary target is to rapidly evaluate, prototype, develop and test Renesas' capacitive touch technology. Besides the tools and software already mentioned here it also contains the Capacitive Touch Workbench for Renesas Synergy™ (CTW), a PC-based tool that aids tuning and optimizing of touch-sensing parameters. Available at the Renesas Gallery from SSP Utilities section, where else?

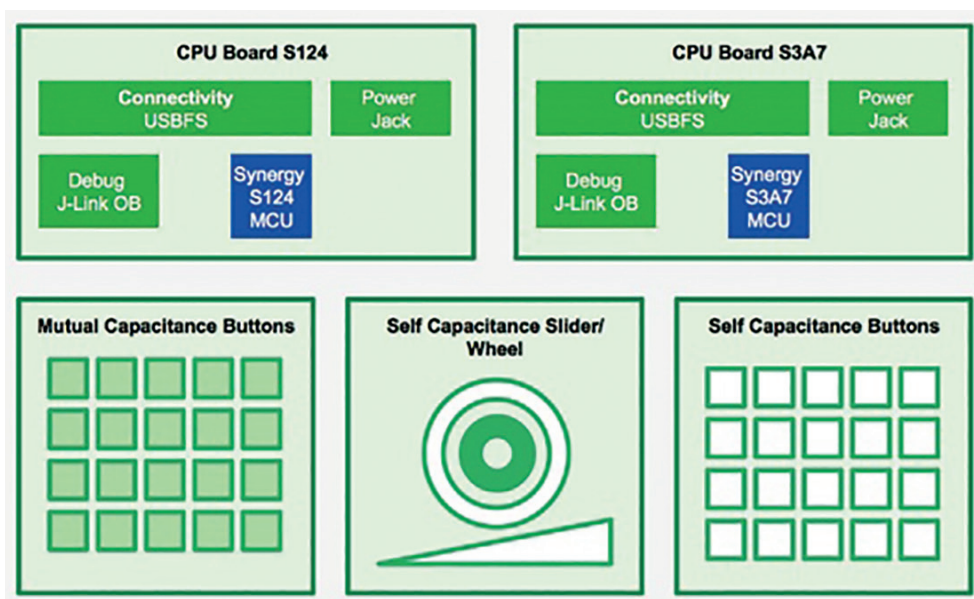


Fig. 110: AE-CAP1 block diagram

The last element of the Synergy Platform is THE entry point for everything related to the Renesas Synergy Platform: the Renesas Synergy Gallery:
<https://synergycastle.renesas.com/>

Just a short registration and the complete Synergy Platform is open for you. It provides quick and easy access to:

- Software
- Downloads
- Tools
- Licenses
- Documentation
- Support
- Examples
- ...

Also available on the Gallery are partner showcases. These projects are developed, offered and supported directly by the Renesas partner. With these projects the partners demonstrate the use of a wide range of software, utilities and tools and offer a quick and easy evaluation of projects to build on your own. The projects may use specific hardware. The CANopen stack for Synergy from port GmbH is an excellent example for a partner showcase. CANopen is a layer 7 software stack on top of CAN, mainly used in industrial automation. The CANopen library from Port implements the communication profile CiA 301 and provides all specified services.

Even the best product may need some support from time to time, in particular a powerful product like the Renesas Synergy Platform. But full service and support is included if you use Synergy:

- Regional support
- Chat with Synergy experts
- Knowledge base
- Resource library
- Renesas Rulz forum
- Synergy Explorer

Just make your own experience with Renesas Synergy – it's waiting to boost your development!

12. LAB

The target of this lab is to give a short and basic introduction into the application development using the Synergy Platform and the benefits it provides. As there are already a lot of documents for Renesas Synergy Platform this introduction is more practical oriented and shows the features of the Synergy Platform. Final target application is a small home automation system using some additional hardware. For this goal many features of the Synergy Platform are used, like HAL (Hardware Abstraction Layer), RTOS (Real-Time Operating System) or connectivity modules. The lab guides you step by step from the very basics to enhanced features to realize the home automation.

The development of the smart home application starts with specifying the needed hardware. Then the BSP is configured to match the hardware specification and provide foundation for the HAL driver modules. After that the HAL is used to read out IO-pins and ADC-values to measure the value ranges of the used sensors. Then the guide makes use of the ADC-Framework working on top of the ADC-HAL driver. It follows the development of a console application for further debug purposes. Thereafter the HAL-driver is used to implement a simple one-wire protocol. Finally the GUI is developed and everything is put together for the final application.

Content of the labs in short:

- Chapter 1:** Figure out the features and needs of a small home automation system
- Chapter 2:** Work with the microcontroller data sheet
- Chapter 3:** Familiarize with the Starter Kits and adapt your home automation system
- Chapter 4:** Introduction to ISDE e²studio
- Chapter 5:** Using the BSP
- Chapter 6:** Using the HAL & RTOS ThreadX[®]
- Chapter 7:** Benefits and features of the application frameworks
- Chapter 8:** How to connect your Starter Kit to your PC using a telnet connection
- Chapter 9:** Develop your touch LCD GUI
- Chapter 10:** Using the message framework to finalize your home automation system

After the lab (and the lecture) you understand the Renesas Synergy Platform and its components, modules and tools. You are able to use the development environment and features to start application development.

As the previous knowledge will be rather unequal the content of the lab can be adapted accordingly. Nevertheless it is recommended to exercise even the basic chapters as they are important for the understanding of the higher level features. Remember: repetition helps a lot to understand.

Even if you do not want to use the optional sensor kit most of the labs are still ready to use. Or get some other sensors instead to realize your application. Keep in mind: Synergy simplifies your life to focus on application.

The lab uses the following components and modules:

- SK-S7G2 Renesas Synergy Starter Kit
- SSP 1.2.1
- e²studio 5.4.0.018
- TraceX[®] 5.2.0
- GUIX Studio 5.3.2.2
- Optional: KOOKYE Smart Home Sensor Kit 16 in 1

Besides the hardware you will need access to the Renesas Synergy Gallery as everything you need you can get at this single entry point to the Synergy world.

You can get your account here:

<https://synergygallery.renesas.com/auth/login>

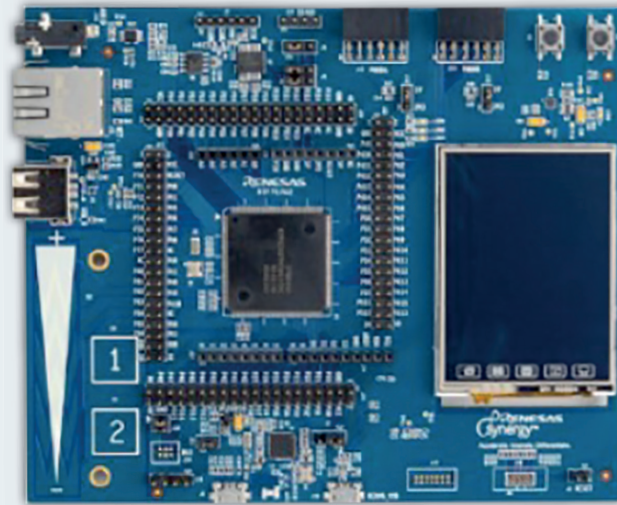


Fig. 111: SK-S7G2 Starter Kit

SMART HOME IOT SENSOR KIT PACKING LIST

No.	Picture	Projects	PCS	No.	Picture	Projects	PCS
1		Voltage Detection Sensor Module	1	10		HC-SR501 Infrared PIR Motion Sensor Module	1
2		MQ-2 Gas Smoke Sensor Module	1	11		Digital Touch Sensor Module	1
3		MQ-5 Combustible Gas Detector Sensor Module	1	12		Photosensitive Light Sensor Module	1
4		MQ-7 CO Carbon Monoxide Detector Sensor Module	1	13		Vibration Sensor Module	1
5		Flame Detection Sensor Module	1	14		Sound Detection Sensor Module	1
6		Water Level Sensor Module	1	15		5V 2-Channel Relay Module	1
7		DS18B20 Temperature Sensor Module	1	16		Buzzer Alarm Sensor Module	1
8		DHT11 Temperature & Humidity Sensor Module	1	17		5 Pin Jumper Wires (20cm, Female to Female)	2
9		BMP180 Digital Barometric Pressure Sensor Module	1	18		Plastic Box	1

Fig. 112: Sensors of the KOOKYE Smart Home Sensor Kit 16 in 1

12.1. INTERNET OF THINGS & INDUSTRY 4.0

This initial lab is used to create an idea for your own IoT/Industry 4.0 application based the SK-S7G2 Starter Kit. No matter whether it is a home automation system or anything else, please find a system you are eager to develop. Depending on your previous knowledge the complexity may vary, but you should focus on the use of the Synergy Platform. Evaluate the application you want to develop with regard to hardware requirements and additional components you need. Draw a schematic of the application to get a first impression of how it might look like.

The example that is described during the lab uses some sensors of the KOOKYE Smart Home Sensor Kit to realize a small smart home application, shown in figure 113. The smart home application is a room condition monitor. The application should monitor the humidity, the temperature, the brightness of the room and the water level of plants. The values will be displayed on the LCD of the starter kit. This application is common among smart home applications and serves as a good example on how to use the Synergy platform.

The room condition monitoring application needs the following sensors of the Smart Home IOT Sensor Kit:

1. DHT11 Temperature & Humidity Sensor (one wire signal)
2. Photosensitive Light Sensor (analog signal)
3. Water level Sensor (analog signal)

If you want to use other sensors – no problem, just use any analog sensor you want instead of the light and water level sensors.

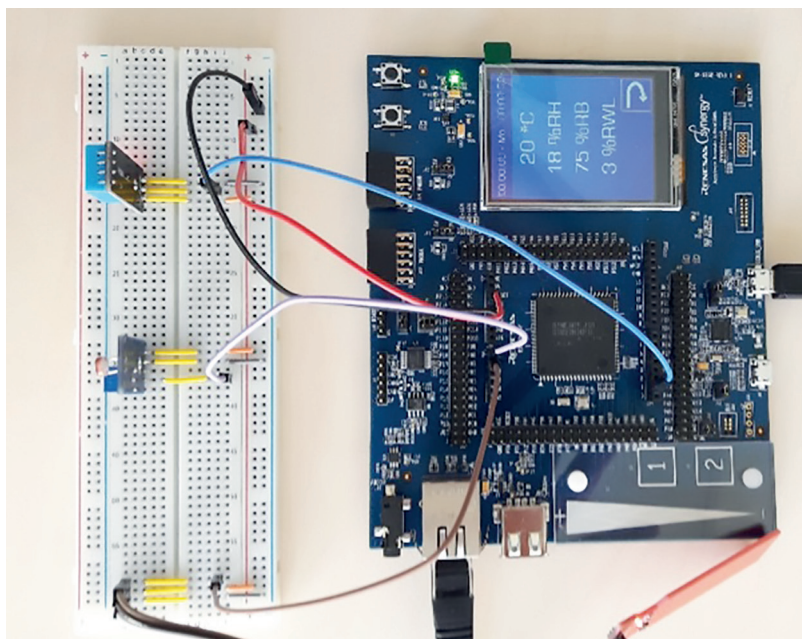


Fig. 113: Smart home application used in this lab

TARGET OF THIS LAB SESSION:

Find a suitable IoT/Industry 4.0 application.

STEPS TO DO:

- Define your application
- Evaluate your application with regard to requirements
- Draw a schematic
- Define what you need, like hardware performance or additional components
- If not done during the preface, get your free account to Renesas Synergy Gallery now (<https://synergygallery.renesas.com/auth/login>)

12.2. MICROCONTROLLER

The heart of each Synergy application is the microcontroller of the Renesas Synergy Family. Four series of MCUs cover a broad range of application areas, from ultra-low power applications like mobile devices to high performance systems. Even though the target of the Synergy Platform is to hide the complexity of the hardware and to enable application development on a very high abstraction level a basic knowledge of the underlying hardware is essential to understand features and limitations. In addition: MCUs are highly sophisticated complex systems and it is really worth looking at these miniaturized engineering marvels!

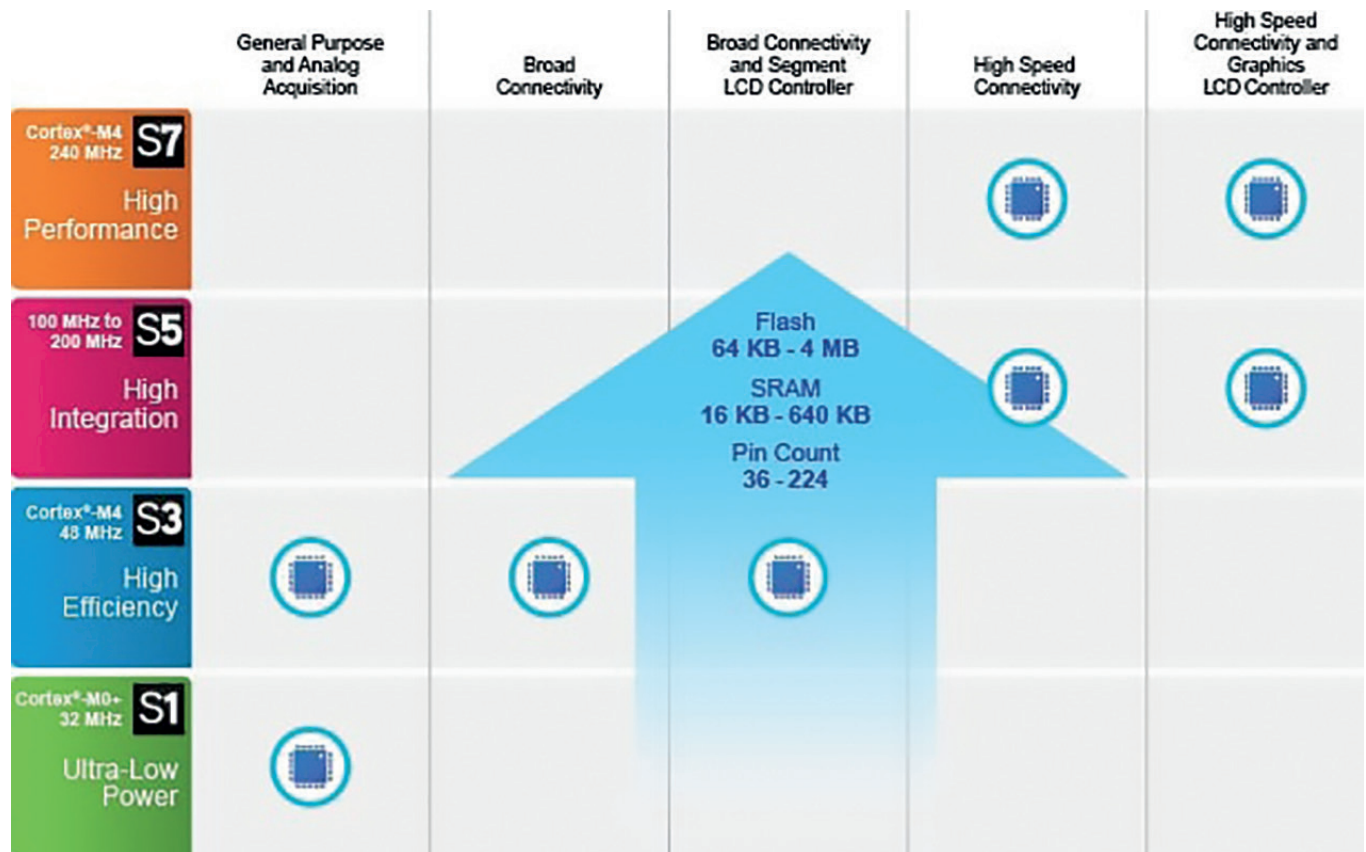


Fig. 114: Synergy MCUs and application areas

The SK-S7G2 Starter Kit uses the S7G2 microcontroller in a 176-pin LQFP package.

TARGET OF THIS LAB SESSION:

Understand and use the MCU data sheet and experience the effort it needs to configure the MCU.

STEPS TO DO:

- Look at the microcontroller Hardware User's Manual ([r01um0001eu0120-synergy-s7g2.pdf](#), just about 2100 pages...) and answer some questions:
 - What is the maximum CPU clock frequency?
 - How many AGT timers are available on this MUC?
 - How many ADC12 channels?
 - Which low power modes are implemented in S7G2?
 - How many non-maskable interrupts are available?
 - What are the absolute maximum rating of the power supply voltage and the operating temperature?
- For the smart home application pin P00 and P01 are used as analog inputs. How can you set these two pins to analog input? And these are just the steps to configure two pins... it is an annoying task, isn't it? So some smart support for this configuration stuff is highly appreciated – so let's use the BSP later on.

12.3. STARTER KITS

The SK-S7G2 is the Starter Kit we will use in this lab. It provides many features to get you started very fast, so no need to be a hardware expert. Nevertheless you should know this Starter Kit to be able to realize your application and maybe connect external components.

TARGET OF THIS LAB SESSION:

Get familiar with the hardware we will use in the lab.

STEPS TO DO:

- **How is the Starter Kit powered?**
- **Find some connections**
 - analog inputs A0 and A1
 - three user LEDs
 - debug USB and Ethernet connector
- **Check whether your application fits to the Starter Kit (use the User's manual, [r12um0004eu0100_synergy_sk_s7g2.pdf](#)), are all connections you need available, get the cables, connectors and all material for your application.**

12.4. ISDE

To start we need the development environment, for this lab we will use Renesas' e²studio. It provides many features to make software and application development as simple as possible. The software is available from the Renesas Synergy web page – where else? You will be guided step by step through the download and installation process and you will perform initial steps in e²studio.

TARGET OF THIS LAB SESSION:

Download and install e²studio 5.2.1.016 and get it running.

STEPS TO DO:

- **Download e²studio from <https://synergygallery.renesas.com/isde>, if computer is behind proxy/VPN, e²studio installer won't be able to download and install GCC ARM Embedded toolchain. In this case, manual installation is required from: <https://launchpad.net/gcc-arm-embedded/4.9/4.9-2015-q3-update>**
- **Install ISDE to your PC/laptop, accept all default installation locations, especially for the GCC compiler; installation instruction are available in the Documentation tab (<https://synergygallery.renesas.com/isde/support#read>) or in "Basics of the Renesas Synergy™ platform" (<https://www.renesas.com/en-eu/products/synergy/book.html>)**
- **Download the SSP from <https://synergygallery.renesas.com/ssp>, installation instruction are available in the Documentation tab (<https://synergygallery.renesas.com/ssp/support#read>) or in "Basics of the Renesas Synergy™ platform"**
- **Get it running and start with a sample project "Blinky"**
 - Connect the Starter Kit to your PC using the DEBUG_USB (J-19) connector - Starter Kit powers up and performs a self-test
 - Touch the LCD touch screen and a pre-programmed thermostat demonstration starts
 - Play around with this application tapping on the symbols of the screen
 - Start the ISDE
 - Select the GCC Arm Embedded toolchain you installed before (if GCC has been installed in the default path)
 - Create a new workspace and name it "room_condition_monitor"
 - You may have to insert your license file which will be located under e2_studio > internal > projectgen > arm > Licenses
 - Create the Synergy Blinky project, which will be the starting point of the application development
 - Name the project SK_S7G2_pincfg, click next and choose in the next window the S7G2 SK board and the SSP version 1.2.1. Click next and choose Blinky with ThreadX then click finish to create the project

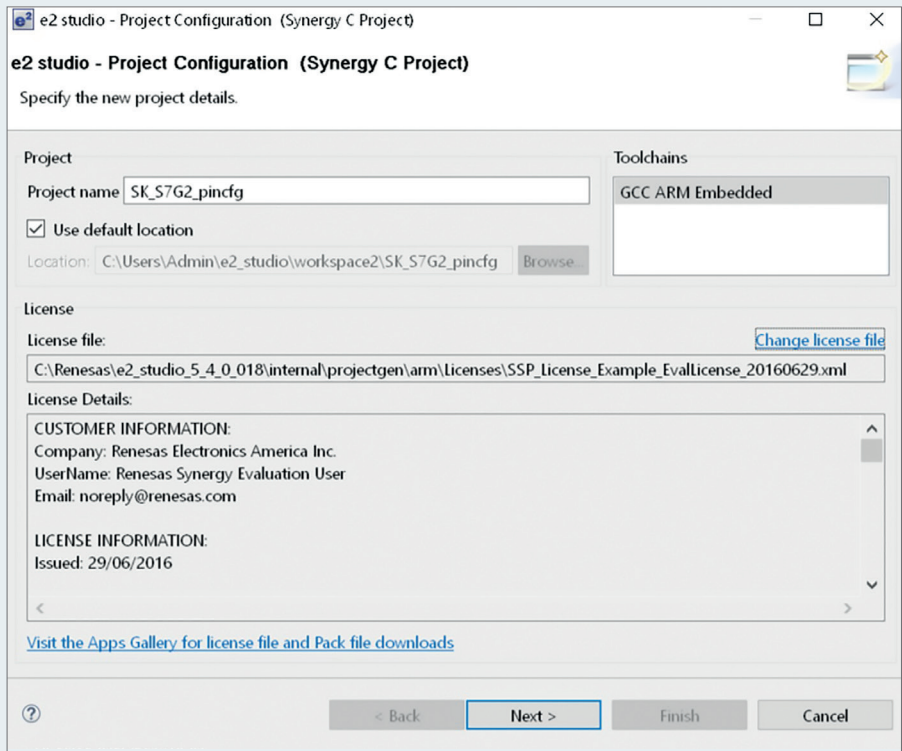


Fig. 115

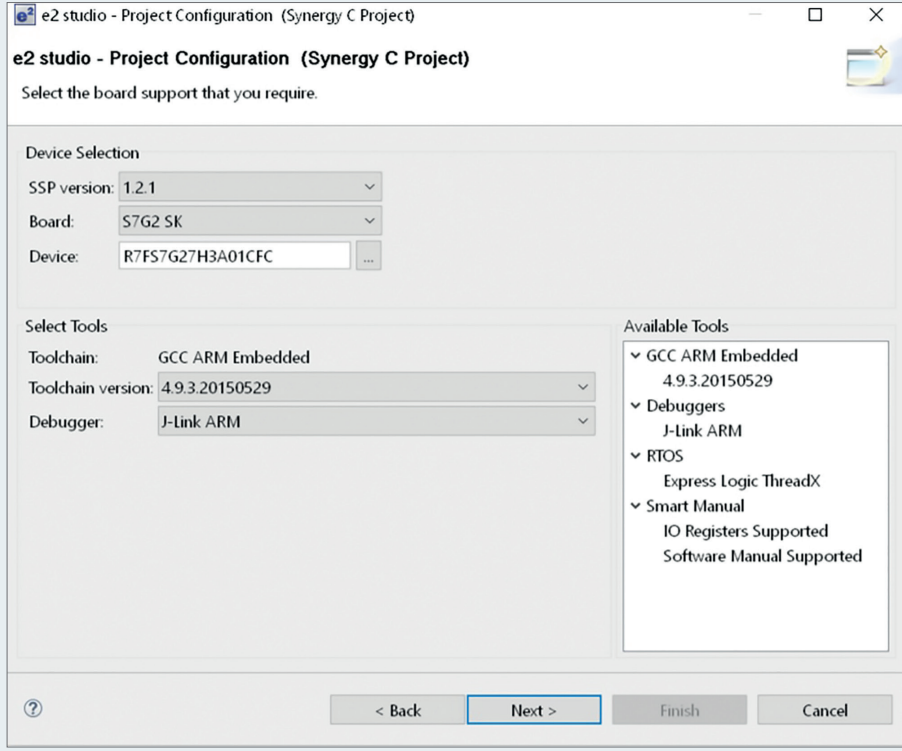


Fig. 116

- The ISDE opens the workspace which is in configuration perspective

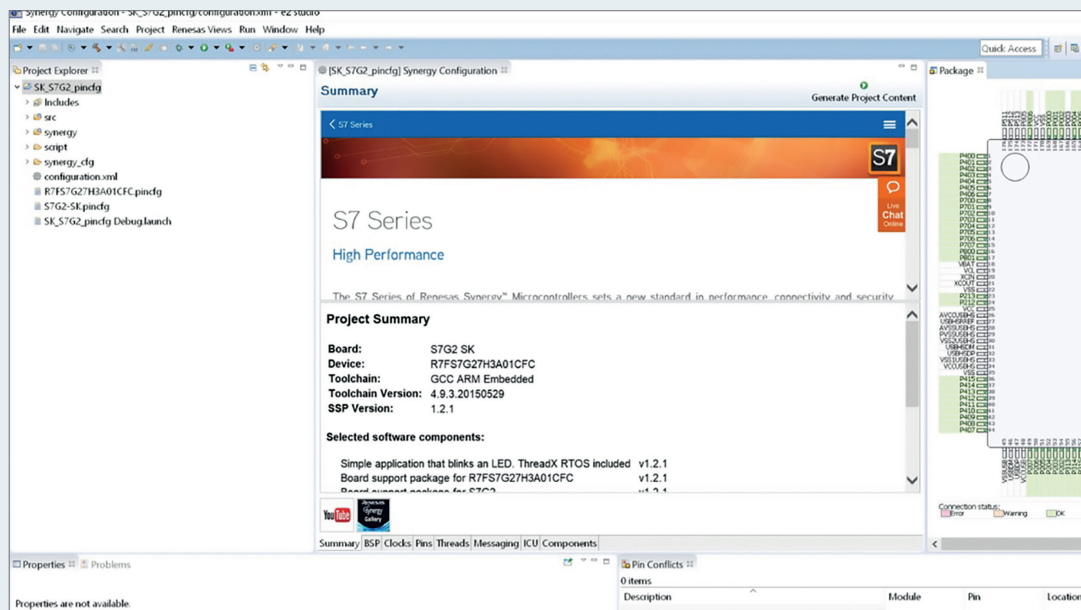


Fig. 117

- Right click on the project folder SK_S7G2_pincfg in the project explorer and start autogeneration of the code by clicking "Generate Project Content". Afterwards build the project by clicking on the small hammer symbol. The compilation should end with 0 warnings and 0 errors.
- Click the "bug button" (small green bug icon) to debug the project.

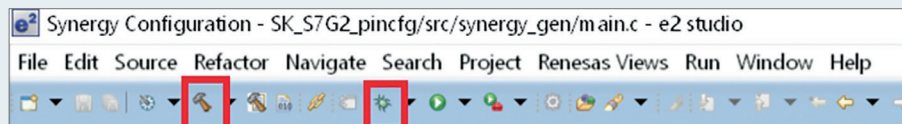


Fig. 118

- Change to the debug perspective by clicking on "Debug" in the top right corner (e2studio will offer switching to "Debug" perspective automatically for every new workspace)
- Run the project by clicking on the green start arrow icon twice (by default the program will stop at the reset vector and at main(), so clicking this button twice is required)



Fig. 119

- The Starter Kit is now flashed with the blinky project. The LEDs should be turning on and off every half second. To stop the debugging process click the red square icon.
- If you encountered any problem during this startup procedure please make sure that everything was installed and connected correctly
- If you disconnect the Starter Kit (switch of the power supply) and reconnect it again the LEDs should blink like before

12.5. BOARD SUPPORT PACKAGE PACKAGE

After the ISDE is running and you started the initial Blinky project let's move towards the configuration of the MCU using the BSP. Now we want to change some basic settings as we want to develop another application – your home automation system. As we will use some external sensors in our example we have to configure the pins accordingly. We will use P000 and P001 as analog input pins for the water level sensor and the photosensitive light sensor and P302 as capture compare input for the DHT11 Temperature & Humidity Sensor. In addition the LCD pins have to be configured as well.

Remember the effort it needs just to configure the two analog pins... annoying... But the BSP makes it much more convenient and reliable.

TARGET OF THIS LAB SESSION:

Configure the pins of the MCU using the configuration tab and the BSP and generate the startup code.

STEPS TO DO:

- Switch to the "Synergy Configuration" perspective of e²studio
- Select the pins tab – here we will configure the three pins P000, P001, P30
- On the left side you see the Pin Selection section: all pins are accessible via the ports or the peripherals
- Analog pins:
 - Click on Ports -> P0 -> P000: as you can see it is already configured as analog input
 - Give the pin the symbolic name brightness as it will be connected to the light sensor (or any other name fitting to your application; as of e²studio 5.4.0.018 the symbolic pin name still does not carry over into source)

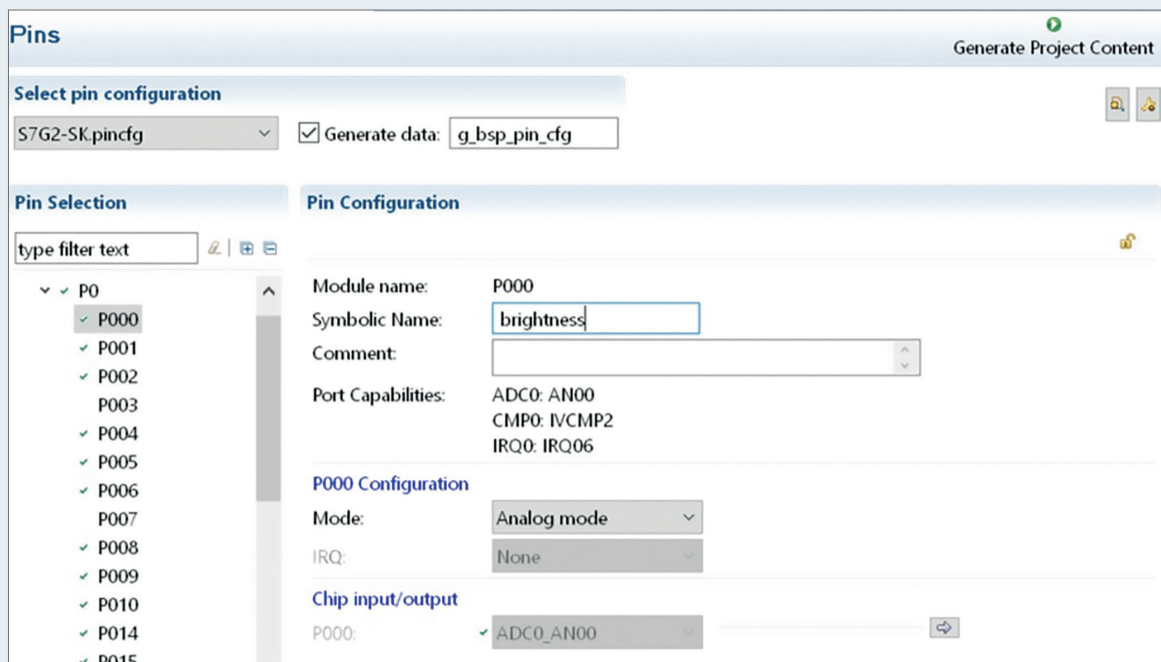


Fig. 120

- Click on Ports -> P0 -> P001: it is also already configured as analog input
- Call it water_level
- Timer pin:
 - Click on Ports -> P3 -> P302: P302 is currently occupied by another module of the MCU
 - Click on the arrow in the lower right corner to navigate to the peripheral module that uses this pin

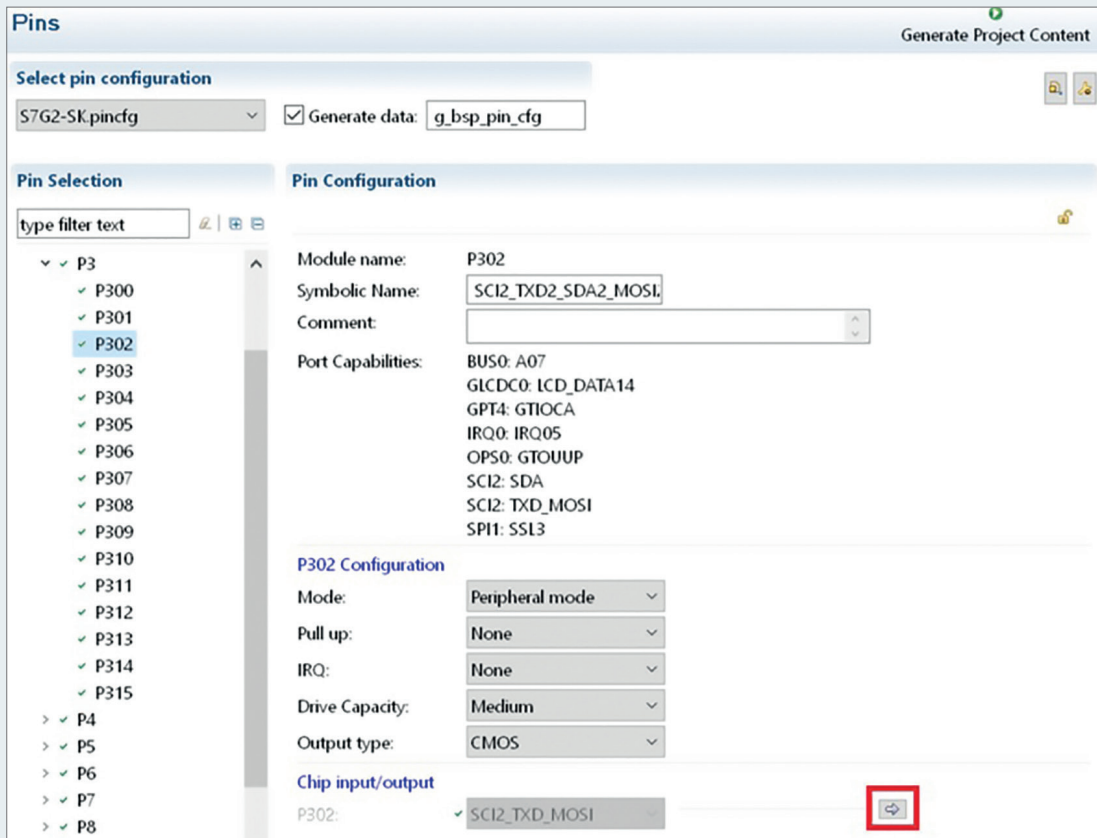


Fig. 121

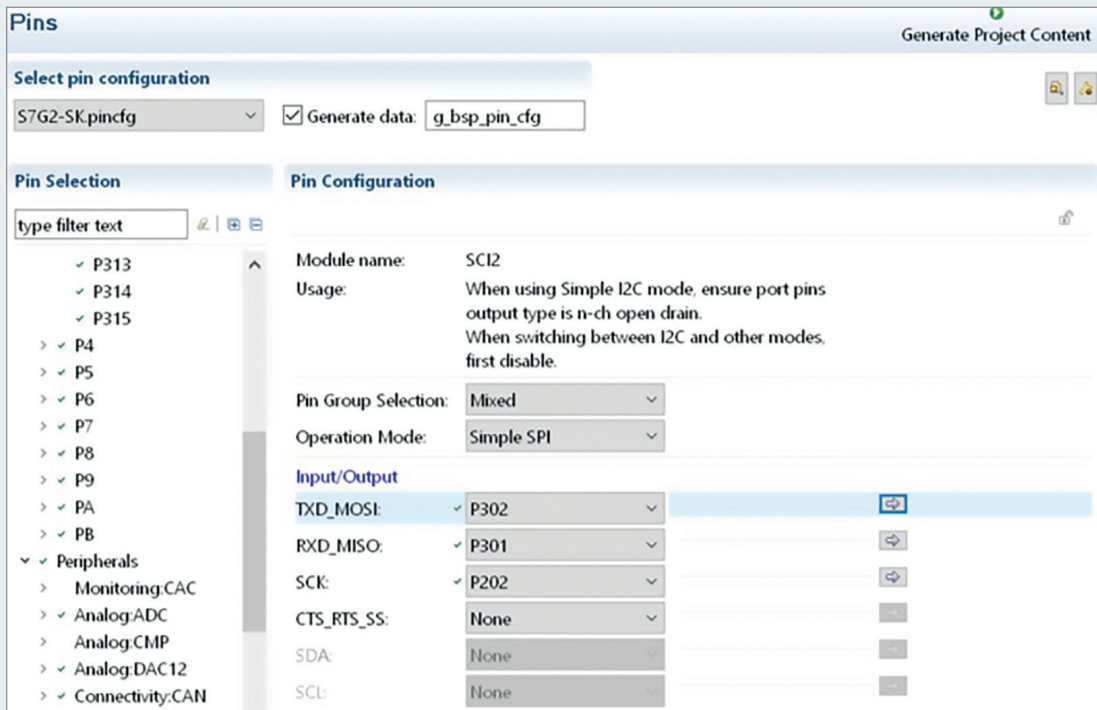


Fig. 122

- The pin P302 is currently used by the SCI module. Disable the module in the Operation Mode selector of the Pin Configuration section. P302 is now unchecked.
- Now set P302 to input mode

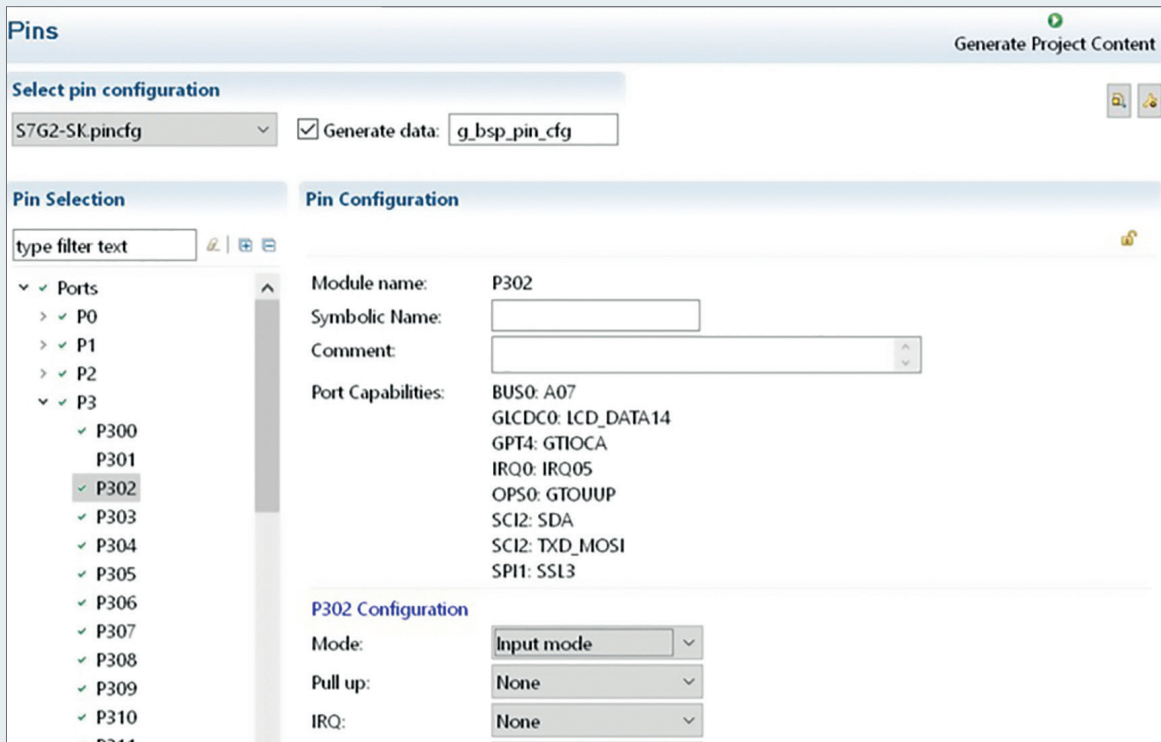


Fig. 123

■ Disable the SPI0 module in Peripherals

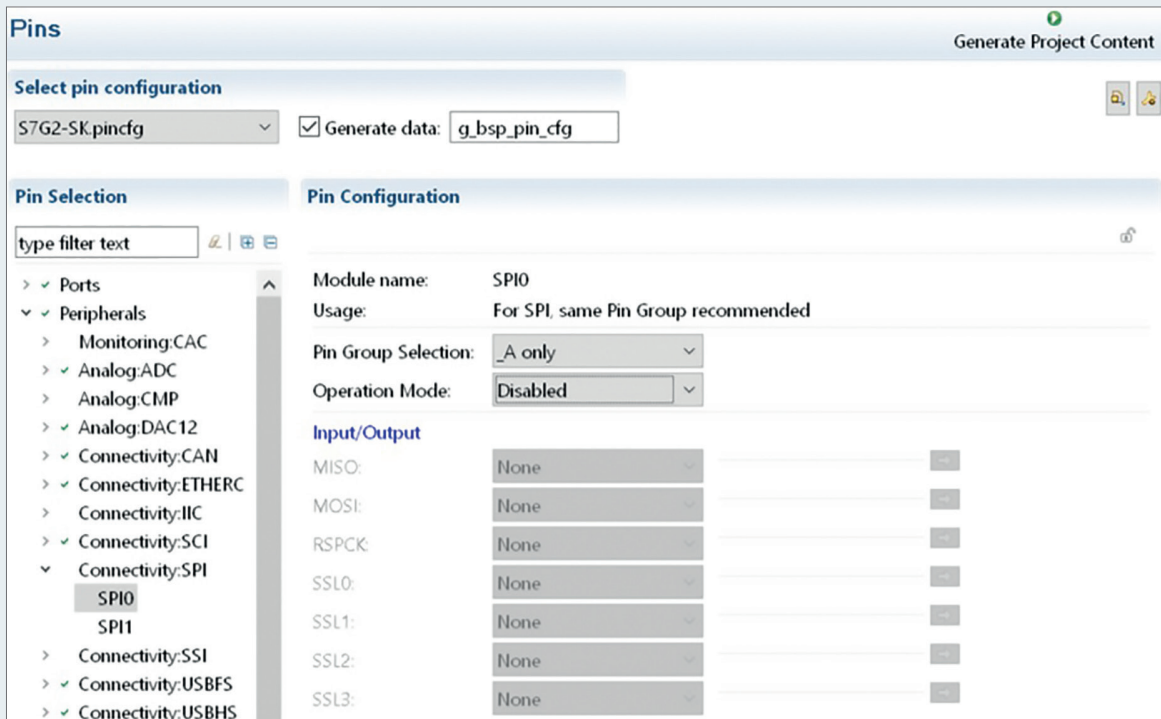


Fig. 124

■ Change the configuration of the SCI0 module

Pins Generate Project Content

Select pin configuration
 S7G2-SK.pincfg Generate data: g_bsp_pin_cfg

Pin Selection type filter text

- > Ports
- > Peripherals
 - > Monitoring:CAC
 - > Analog:ADC
 - > Analog:CMP
 - > Analog:DAC12
 - > Connectivity:CAN
 - > Connectivity:ETHERC
 - > Connectivity:IIC
 - > Connectivity:SCI
 - ✓ SCI0
 - SCI1
 - SCI2
 - ✓ SCI3
 - SCI4
 - SCI5
 - ✓ SCI6

Pin Configuration

Module name: SCI0
 Usage: When using Simple I2C mode, ensure port pins output type is n-ch open drain. When switching between I2C and other modes, first disable.

Pin Group Selection: Mixed
 Operation Mode: Simple SPI

Input/Output

TXD_MOSI: ✓ P101
 RXD_MISO: ✓ P100
 SCK: ✓ P102
 CTS_RTS_SS: ✓ P103
 SDA: None
 SCL: None

Fig. 125

■ Configure the IIC2

Pins Generate Project Content

Select pin configuration
 S7G2-SK.pincfg Generate data: g_bsp_pin_cfg

Pin Selection type filter text

- > Ports
- > Peripherals
 - > Monitoring:CAC
 - > Analog:ADC
 - > Analog:CMP
 - > Analog:DAC12
 - > Connectivity:CAN
 - > Connectivity:ETHERC
 - > Connectivity:IIC
 - IIC0
 - IIC1
 - ✓ IIC2
 - > Connectivity:SCI

Pin Configuration

Module name: IIC2
 Usage: For IIC, use same Pin Group for SDA/SCL signals -Please refer to the MCU User's Manual.

Pin Group Selection: _A only
 Operation Mode: Enabled

Input/Output

SDA: ✓ P511
 SCL: ✓ P512

Fig. 126

■ **Configure the LCD pins:**

- Go to pin Ports > P1 > P115 and select Output mode (Initial High). As this pin drives the LCD write and read signal rename the symbolic name to LCD_WR

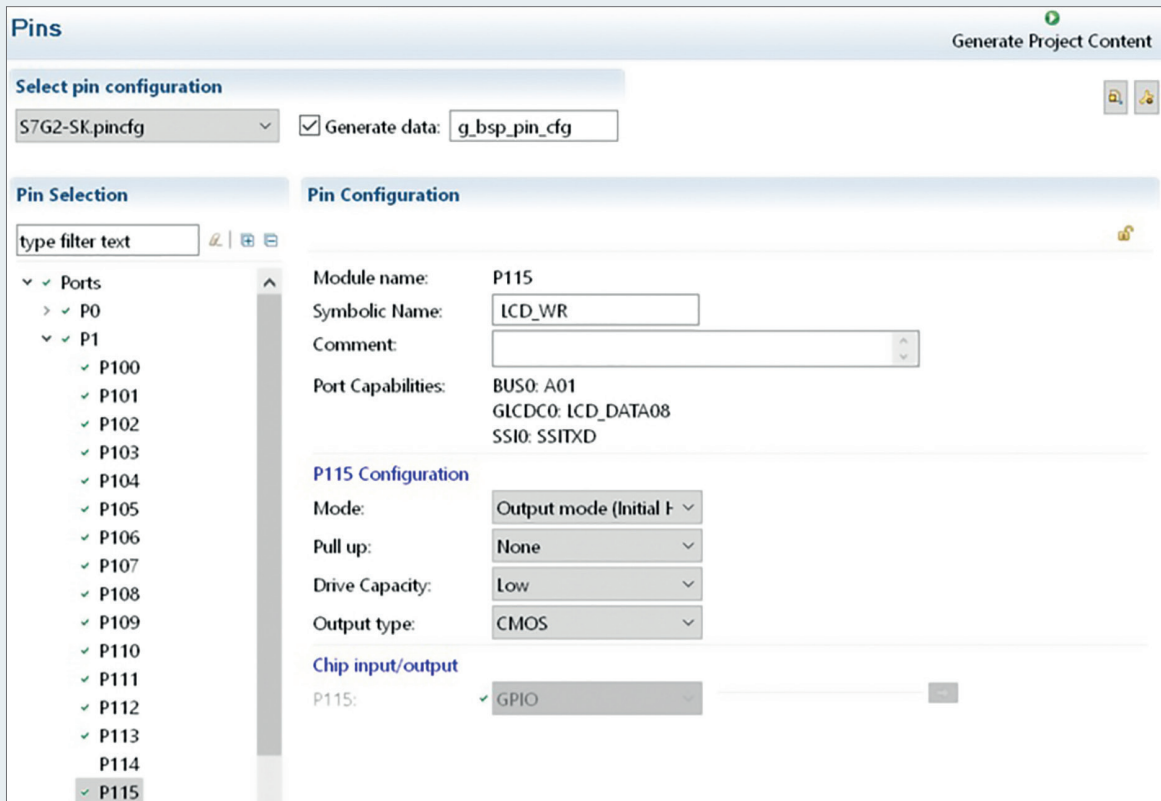


Fig. 127

- Go to Ports > P6 and select Output mode (Initial High) for P609, P610 and P611 as well. P609 is the reset signal for the touch panel, so call it RESET# for the symbolic name. P610 is the LCD reset, name it LCD_RESET. Name P611 LCD_CS as it is the chip select for the LCD.

■ **Configure the LCD pins:**

- Go to pin Ports > P1 > P115 and select Output mode (Initial High). As this pin drives the LCD write and read signal rename the symbolic name to LCD_WR
- Select Graphics: GLCDC > GLCDC0. The configurations are already set by the SK-S7G2 BSP and need no change.

■ **The pin configuration is now complete – just by working with the graphical interface of the configuration view. Rather simple and fast, isn't it?**

- **To check your pins have a look at the package tab. Choose symbolic names in the drop down menu at the top right corner to show the pin names. Any error will be marked in red color.**

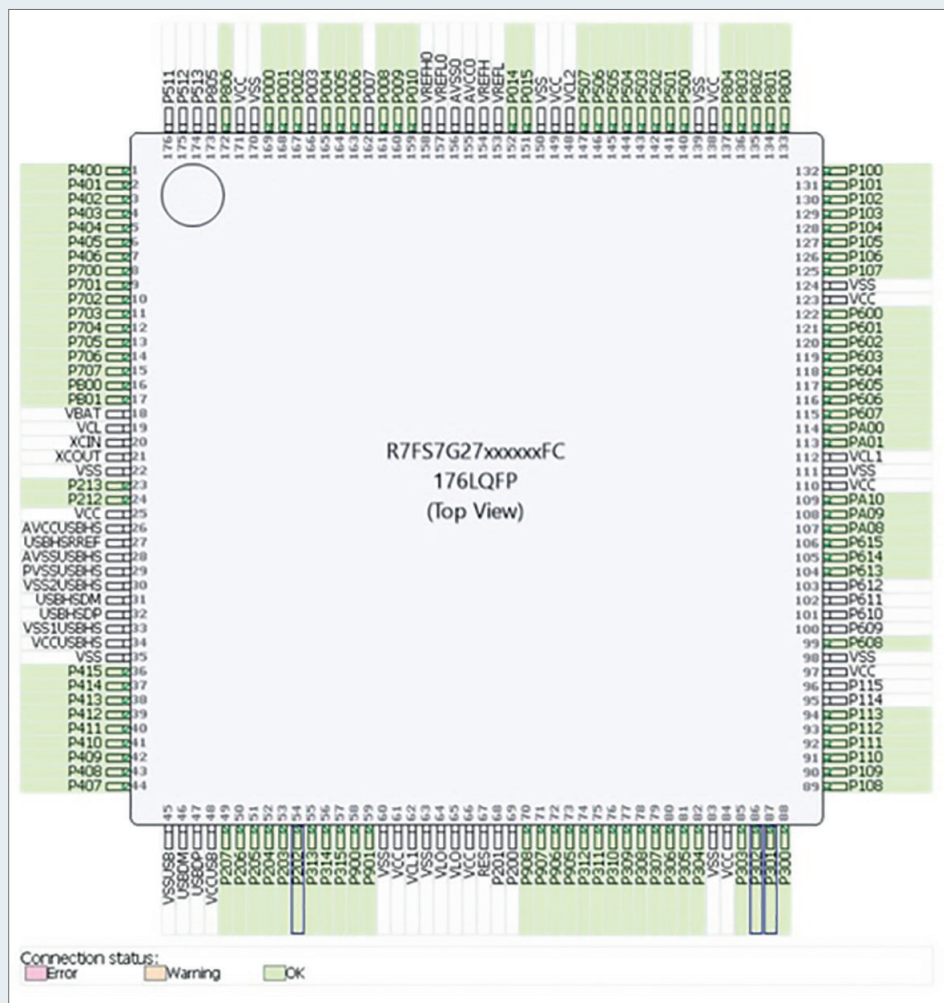


Fig. 128

- Save the configuration
- Build your project, it should end with 0 errors and 0 warnings. The new code is generated, keep in mind that by each build process the code will be generated again
- We will use this project as a template for subsequent projects, so we would like to export it:
 - Right click on the SK_G2S7_pincfg project in the project explorer and select Export Synergy Project. Save the file in the workspace folder as SK_S7G2_pincfg.zip

NOTE:

The purpose of this BSP approach is to auto generate the startup code. Therefore be careful to edit the correct files with your user code later on... The project Explorer on the left side in configuration view lists all currently included files. Within the src folder you can find the synergy_gen folder containing files automatically generated by the Generate Project Content command. These files will be updated by each project generation process, so do not edit these files. For example the pin configuration in pin_data.c. Please play around with the pin configuration and generate code after some changes (and come back to our configuration in the end...). Observe the changes in pin_data.c reflecting the changes you have done in the pin configuration. This folder also contains the famous main.c file. Even this file is generated automatically, so do not edit, all changes will be lost after the next project generation command (try it...). User code will be written in files like hal_entry.c which are located directly in the root of the src folder.

12.6. HARDWARE ABSTRACTION LAYER & RTOS

After our first steps with the BSP let's get started with the hardware abstraction layer and the RTOS. We will create a new project just to read sensor values using the HAL of the SSP starting with the ADC driver to read the analog sensors. We will use a RTOS thread to read the analog inputs periodically. Afterwards we will use the AGT driver to implement the one-wire transmission protocol of the DTH11 humidity & temperature sensor.

TARGET OF THIS LAB SESSION:

Use the HAL drivers and RTOS threads for a first program. Setup the ADC driver using the HAL to convert the analog inputs periodically and write your first lines of code using the HAL API. Afterwards use the HAL-Timer for the DTH11 temperature and humidity sensors.

STEPS TO DO FOR THE ADC DRIVER:

- Attach the sensors to the corresponding pins of the board (A0 and A1)
- Close all other projects
- Import the .zip file created in the BSP lab and rename it to `hal_read_sensors`
 - Right click the project explorer and choose Import > General > Rename & Import Existing C/C++ Project into Workspace
 - Select the SK_S7G2.zip file in the workspace folder
- Test the new project by generating and building the code.
- Debug the project, it should behave like the last project...
- Open the thread tab in the Configuration window. Here we can add new threads. Currently there is just the "Blinky Thread" and the HAL/Common platform containing the drivers for the I/O port driver, the Clock Generation Controller (CGC) and the Factory MCU Information module (FMI)
- Create a new thread by clicking on the add thread icon. The properties window should open itself. Rename the thread to `hal_adc_thd` and leave the remaining configuration as is.
- Add an ADC driver by clicking on the add icon on the `hal_adc_thd` window. Select Driver > Analog > ADC Driver on `r_adc`. The ADC driver is now added to the thread. Like the ADC driver all Synergy HAL drivers are suitable for use in the RTOS environment.

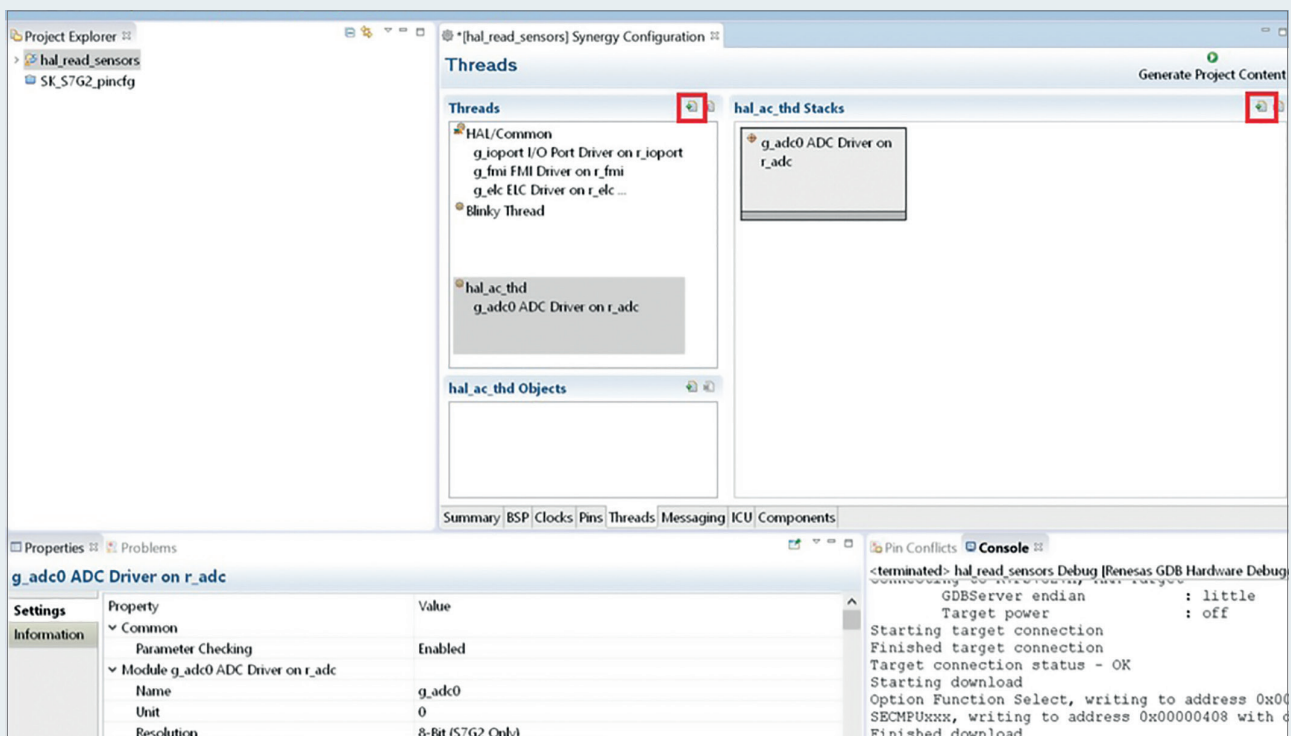


Fig. 129

- Click on the adc stack and have a look at the properties tab (lower left corner). Here we will configure the ADC to fit to our application
- Setup of ADC channels

- Have a look at the Port Capabilities of the ADC pins P000 and P001 (in Pins tab) and figure out which ADC channels need to be configured. P000 is associated with ADC channel 0 and P001 with ADC channel 1.

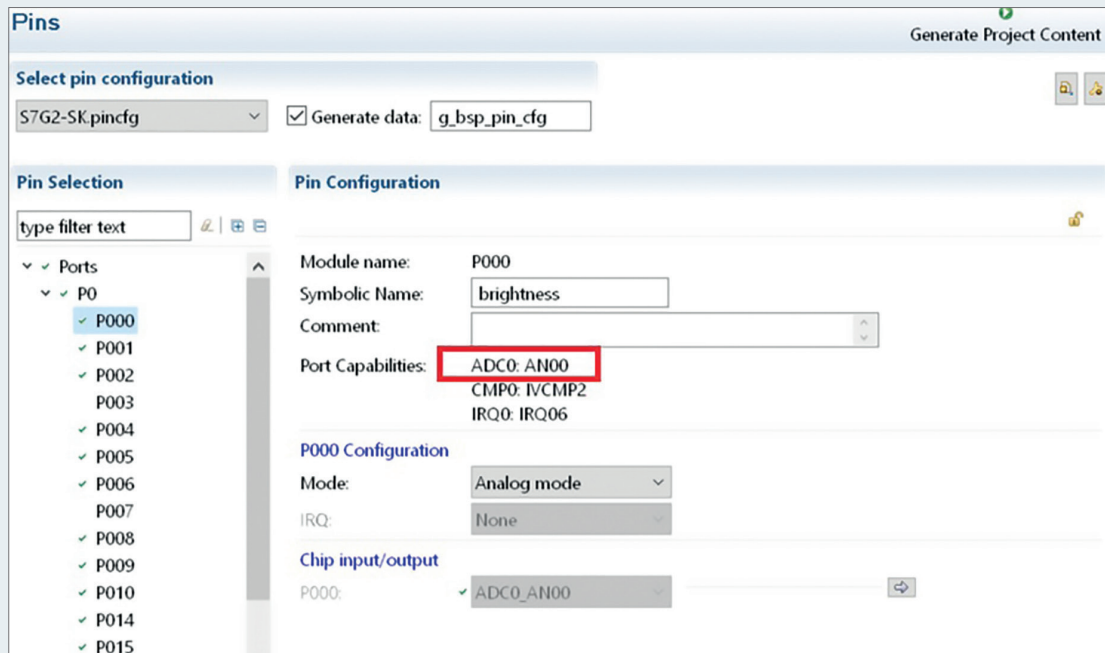


Fig. 130

- Return to the properties tab of the ADC stack and enable channel 0 and 1 in Normal Mode under Channel Scan Mask

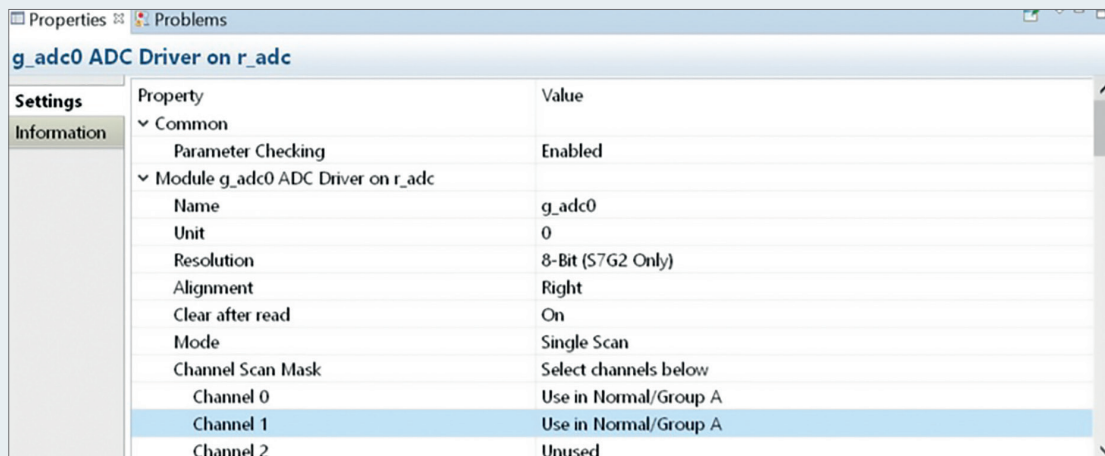


Fig. 131

- Scroll down the properties tab to Addition/Averaging Mask and enable the mask for channel 0 and 1. Under Add / Average Count select Average two samples. Keep the remaining configurations.

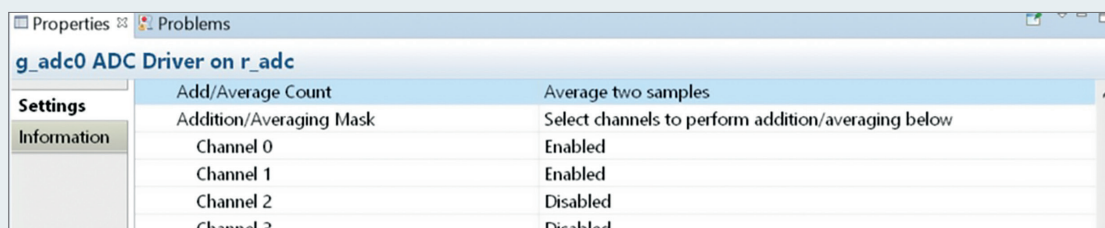


Fig. 132

- The setup of the ADC is now done – again rather simple... Save the configuration, generate the project and switch to C/C++ perspective.
- As you can see, besides the files you already know from last lab, there is an additional file in the src folder: hal_adc_thd_entry.c. This file (which is not updated after next generate project process) will contain the user code associated with the hal_adc_thd thread. So at this point we can start to program user code.
- User code for hal_adc_thd thread
 - Inside hal_adc_thd_entry.c replace the dummy function void hal_adc_thd_entry(void) with the code listed below:

```

#include "hal_adc_thd.h"

/* hal_adc_thd entry function */
void hal_adc_thd_entry (void)
{
    /* error variables */
    ssp_err_t err;

    /* variables for the sensor values */
    adc_data_size_t water_level_adc_t = 0;
    adc_data_size_t light_sensor_adc_t = 0;

    /* opens the adc hal driver */
    err = g_adc0.p_api->open(g_adc0.p_ctrl, g_adc0.p_cfg);

    /* catch error */
    if(err != SSP_SUCCESS)for(;;);

    /* loads the adc configuration */
    err = g_adc0.p_api->scanCfg(g_adc0.p_ctrl, g_adc0.p_channel_cfg);

    if(err != SSP_SUCCESS)for(;;);

    /* don't use a thread without an endless loop */
    while (1)
    {
        /* start the adc scan */
        err = g_adc0.p_api->scanStart(g_adc0.p_ctrl);
        if(err != SSP_SUCCESS)for(;;);

        /* polling for ready status */
        while(SSP_ERR_IN_USE == g_adc0.p_api->scanStatusGet(g_adc0.p_ctrl));
        if(err != SSP_SUCCESS)for(;;);

        /*read adc channel 0 value */
        err = g_adc0.p_api->read(g_adc0.p_ctrl, ADC_REG_CHANNEL_0,
        &light_sensor_adc_t);
        if(err != SSP_SUCCESS)for(;;);

        /*read adc channel 1 value */
        err = g_adc0.p_api->read(g_adc0.p_ctrl, ADC_REG_CHANNEL_1,
        &water_level_adc_t);
        if(err != SSP_SUCCESS)for(;;);

        /*put this thread to sleep for 1 RTOS timer ticks - 1 tick =10ms */
        tx_thread_sleep (1);
    }
}

```

- **This user code can be divided into two parts: initialization and the working loop**
 - Initialization: the thread initializes two variables (`water_level_adc_t`, `light_sensor_adc_t`) for the ADC values. Afterwards the ADC driver is opened and the configuration is loaded
 - Working loop: the thread starts the ADC scans, waits for the scans to be finished and reads the converted values of channel 0 and 1. Afterwards it waits for 1 tick and then restarts
- **All functions you need are accessed with the `g_adc0` instance. This object represents an instance using configuration, control and API structures. All of these are referenced as pointers so the instance does not contain them. Config and control structures are usually specific to the particular instance, but API structure is generic and is taken from the driver code. You can use the autocomplete function to get the available options:**
 - Type the stack name `g_adc0`. and press “Strg+[Space]” (autocomplete): the available options are shown
 - This functionality and object oriented style of programming is available throughout all driver and Framework modules – which simplifies programming a lot!
- **Save and build the project. Debug it and change to debug perspective**
- **Set a breakpoint by double clicking to the line number where the breakpoint should be set, e.g. the line “`tx_thread_sleep (1);`”. The program will stop at the breakpoint after resuming the program.**
- **The variables are displayed at the top right corner. ave and build the project. Debug it and change to debug perspective**
- **Play around with the sensors (vary the light for the light sensor, put the water level sensor into a glass of water**
- **Try to determine the maximal and minimal values converted by the ADC (as the ADC is set to 8 Bit resolution the range will be 0 to 255**
- **If you want to change the ADC resolution: it’s very simple as you just have to change the configuration**
 - Stop debugging
 - Switch to the properties tab of the ADC driver and select your Resolution (e.g. 12 bit)
 - Save and generate the project, debug it, ... see above.

STEPS TO DO FOR THE TIMER:

- **The DTH11 sensor requires a simple input pin (and the power supply) in order to implement the one-wire protocol. Attach the DTH11 sensor to the corresponding pin P302.**
- **The pin P302 was already configured before. Check that the configurations are correct.**
- **Add a new thread and call it `hal_timer_thd`. The thread configuration does not need any change.**
- **The thread should use the asynchronous general purpose timer module (`agt-timer`). Add a new software stack to the timer thread under “Driver>Timers>Timer Driver on `r_agt`”.**

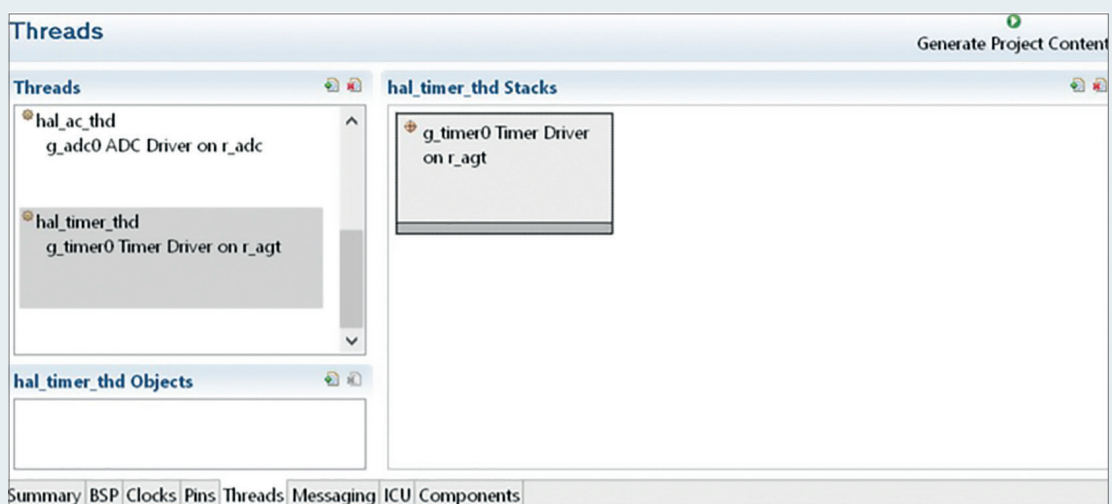


Fig. 133

- Open the properties window next and set the timer stack configuration as shown below:

Property	Value
Common	
Parameter Checking	Default (BSP)
Module g_timer0 Timer Driver on r_agt	
Name	g_timer0
Channel	1
Mode	Periodic
Period Value	1
Period Unit	Microseconds
Auto Start	True
Count Source	PCLKB
AGTO Output Enabled	False
AGTIO Output Enabled	False
Output Inverted	False
Callback	timer0_counter_callback
Interrupt Priority	Priority 0 (highest)

Fig. 134

- The agt timer stack offers 2 channels, channel 1 is used here but channel 0 can be used as well. The timer runs periodically after start (the counter keeps running after the underflow) and the underflow-time is set to 1 μ s. This time equates 240 instructions with a CPU frequency of 240 MHz.
- The DHT11 one-wire protocol take approximately 3.4 ms. Within this time we need to initiate a start sequence and read 40 bits of data. To protocol uses high level pulses of different width from 24 μ s to 80 μ s. Therefore the 1 μ s resolution is sufficient for this protocol.
- Add a callback function "timer0_counter_callback". This function will be called from the timer interrupt routine whenever the timer overflows.
- The interrupt priority must be higher than the lowest priority setting as this setting is reserved for SysTick timer.
- The next step is to figure out how the detailed timing schemes for the DHT11 one-wire protocol. The protocol is defined in the data sheet of the DHT11.

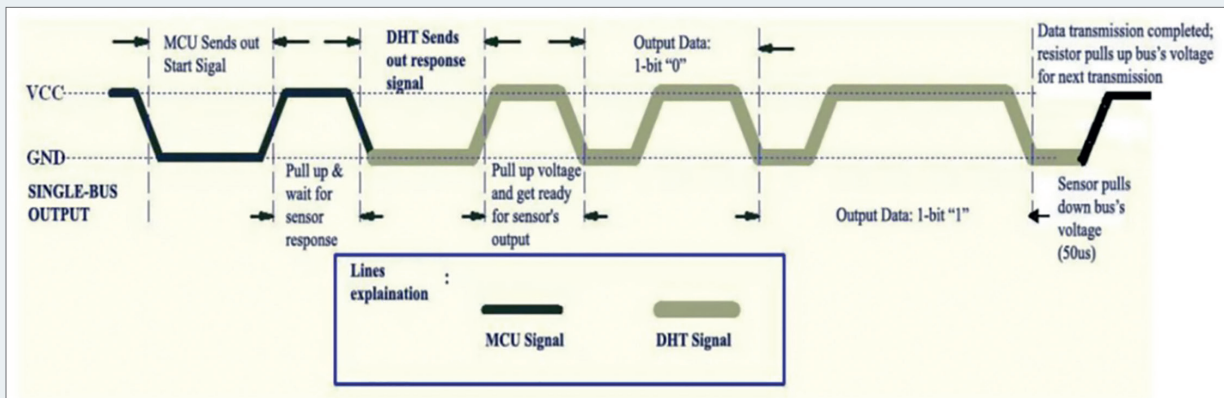


Fig. 135

- The first important step is to get the sensor to transmit data so that it can be checked with an oscilloscope. Therefore the start sequence has to be programmed first. According to the DHT11 datasheet the data transmission is always initiated by the master device.

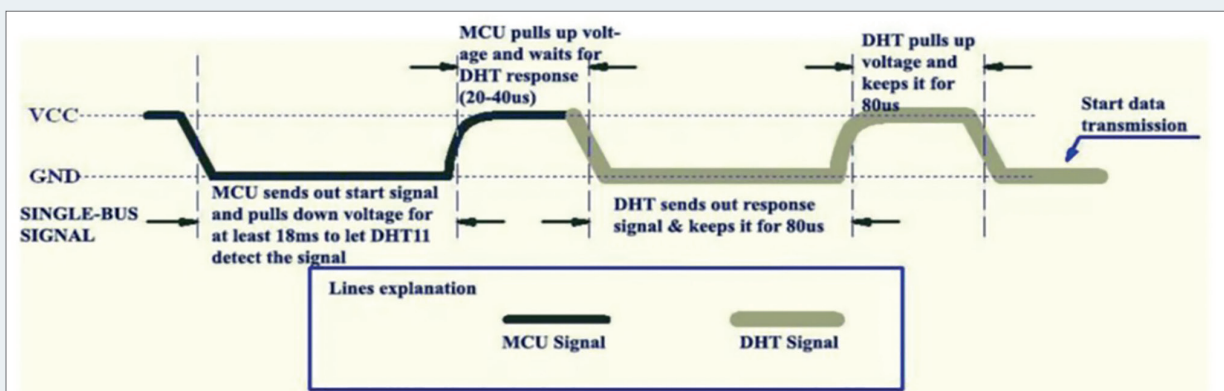


Fig. 136

- The master start signal consists of a low level signal with a minimum width of 18 ms. After the start signal is given the master switches to input mode to receive the DTH11 data. Note: the DTH11 needs one second of time for full functionality after powering up.
- The DTH11 answers with a slave start signal of its own as illustrated below.
- The DTH11 sends the data bits after the slave start bits. A "0" bit is indicated by a 26 μ s high level pulse. A "1" bit uses a 70 μ s pulse. The DTH11 sends a total of 5 bytes.

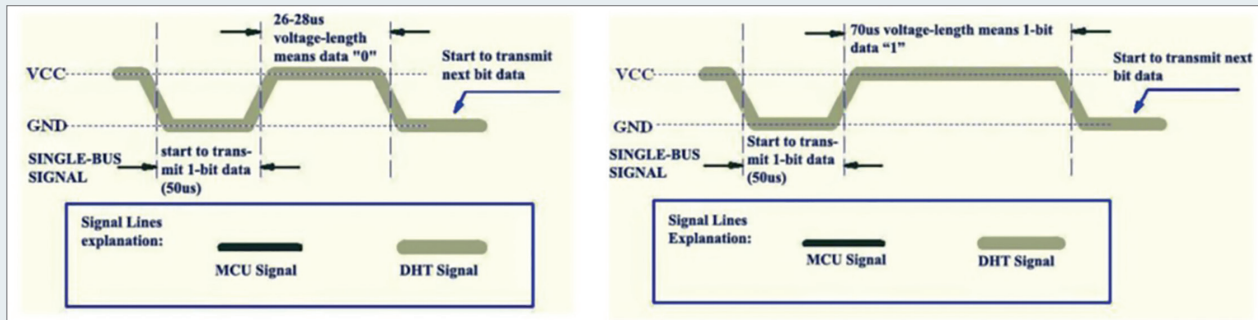


Fig. 137

■ Programming the dth11 protocol

- The code for giving the start signal is put into a separate function, which is posted below. The function initiates the DTH11 to start the transmission process.

```

ssp_err_t dth11_master_start_signal(void)
{
    ssp_err_t err;

    /* output level low */
    err = g_adc0.p_api->pinWrite(IOPORT_PORT_03_PIN_02,
                                IOPORT_LEVEL_LOW);

    /* set mode to output */
    err = g_ioport.p_api->pinDirectionSet(IOPORT_PORT_03_PIN_02,
                                           IOPORT_DIRECTION_OUTPUT);

    /* wait 20ms */
    tx_thread_sleep(2)

    /* set mode to input */
    err = g_ioport.p_api->pinDirectionSet(IOPORT_PORT_03_PIN_02,
                                           IOPORT_DIRECTION_INPUT);

    /* delay for rising level */
    R_BSP_SoftwareDelay(BSP_DELAY_UNITS_MICROSECONDS,
                        30);

    /* check if sensor pulled level low */
    g_ioport.p_api->pinRead(IOPORT_PORT_03_PIN_02,
                           &P302_level);

    /* if level high then abort protocol */
    if(P302_level == IOPORT_LEVEL_HIGH)
    {
        err = SSP_ERR_ABORTED;
    }

    /* return status */
    return err;
}

```

- The function uses the ThreadX wait function `tx_thread_sleep(2)` to implement the necessary 18 ms low level. To be exact the wait function waits 20 ms, since the implemented tick step in the SSP is 10 ms.
- The pin P302 is accessed via the I/O Port Driver. The HAL/Common section allows all software stacks to be accessed by other threads.
- The ports can be accessed with the `g_ioport` object like shown in the implementation above. In the first section the pin P302 is configured as output and the level set low. After the waiting time of 20ms the mode is changed to input which releases the low level. A callback function "timer0_counter_callback". This function will be called from the timer interrupt routine whenever the timer overflows.
- waiting time of 30 μ s is implemented with the BSP delay function which is used alternatively The following to a timer. The DTH11 should pull the level down within 30 μ s, if this is not the case then we assume that an error occurred.
- The illustration below depicts an oscilloscope screenshot of the DTH11 protocol using the master start signal function. Note: the DTH11 sensor should not be read faster than once per second.

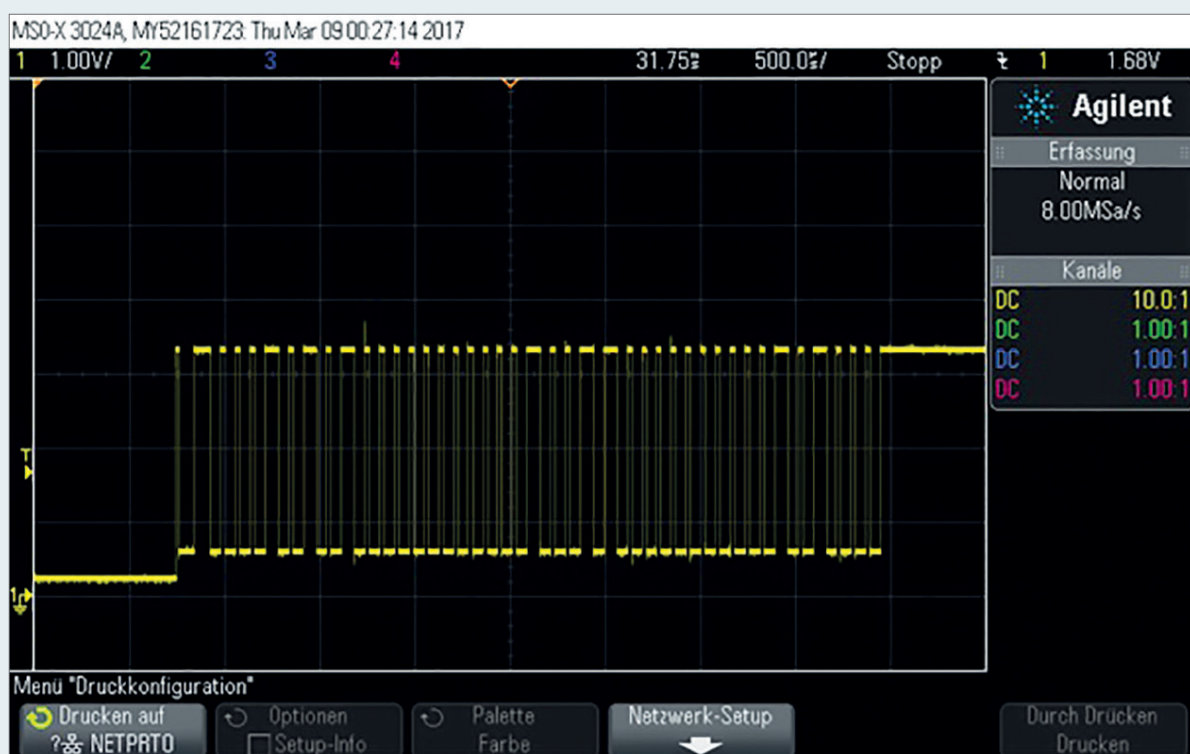


Fig. 138

- It is important to verify the widths of the individual signals. The slave start high level signal is used to synchronise the master to the slave for reading the data bits. The width of the high level signal is approximately 86 μ s. Note: that the width of the signals may differ from sensor to sensor since the low cost sensor is not the most accurate.

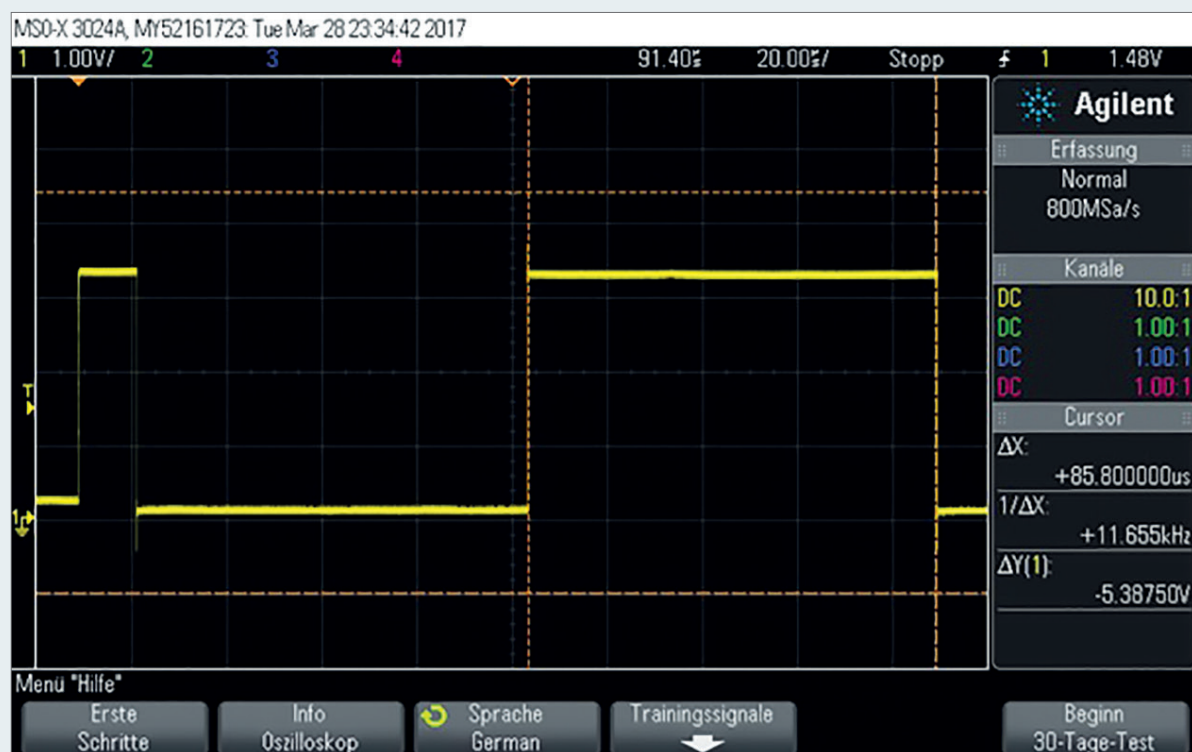


Fig. 139

- The width of the "0" and "1" bit signal are 24 μ and 70 μ respectively.

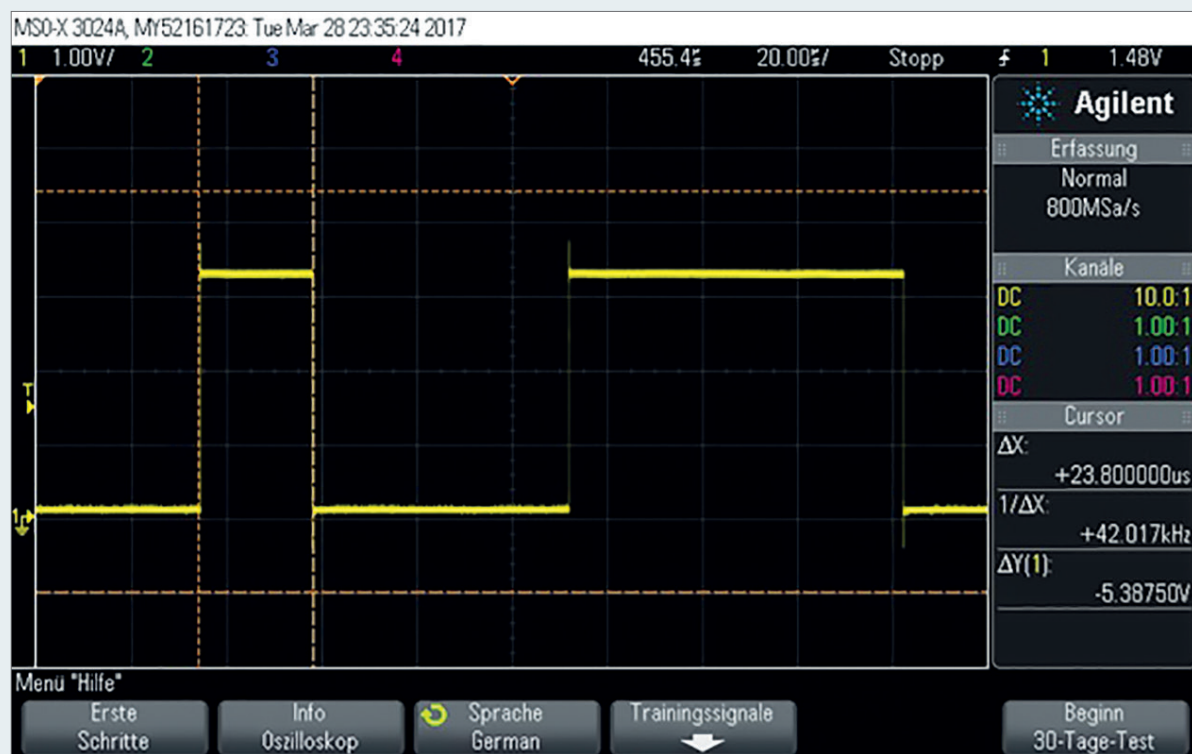


Fig. 140

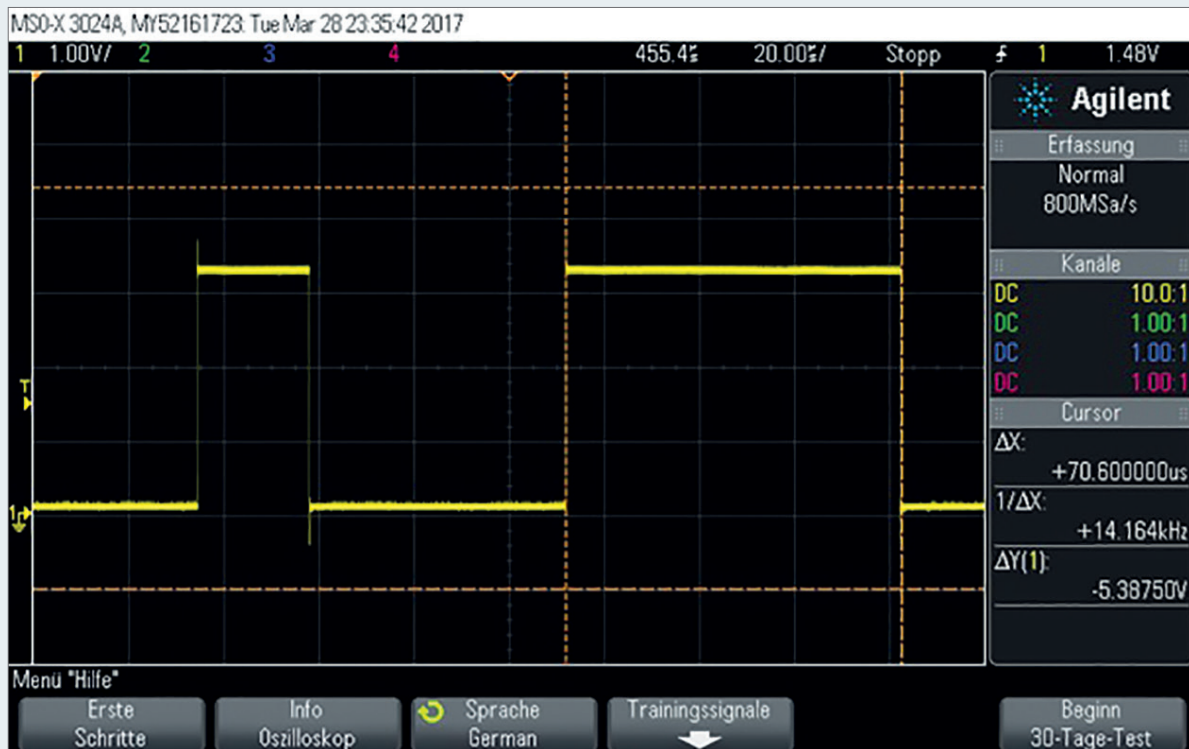


Fig. 141

- The protocol functions that are implemented are shown below.

```

ssp_err_t dth11_protocol_init(void);
ssp_err_t dth11_protocol(uint8_t* const humidity,uint8_t* const temperature);
ssp_err_t dth11_master_start_signal(void);
ssp_err_t dth11_sync_slave_start_signal(void);
ssp_err_t dth11_get_slave_data_bits(uint8_t* const data_bit);
ssp_err_t dth11_check_and_asign_data(uint8_t* const humidity,uint8_t* const temperature);

```

- The protocol init function initializes the timer.

```

ssp_err_t dth11_protocol_init(void){
    ssp_err_t err;
    err = g_timer0.p_api->open(g_timer0.p_ctrl,
                               g_timer0.p_cfg);
    return err;
}

```

- The protocol function manages the DTH11 protocol. It starts with the master start signal which is followed by the slave start signal. After a successful synchronisation the 40 bits are read with the help of the get data bits functions. The dth11_data[] array is a static uint8_t variable that is used to store the bits and save them as bytes. The last function is meant to check and assign the data. The DTH11 uses an 8 bit checksum to verify the success of the transmission that the function checks against.


```

ssp_err_t dth11_protocol(uint8_t* const humidity, uint8_t* const temperature)
{
    ssp_err_t err;
    uint8_t data_bit;

    err = SSP_SUCCESS;

    /* master initiates transmission */
    err = dth11_master_start_signal();
    if(SSP_SUCCESS == err)
    {
        /* sync on slave start signal */
        err = dth11_sync_slave_start_signal();
        if(SSP_SUCCESS == err)
        {
            /* total of 5 bytes to receive */
            for (int i = 0; i < 5; ++i)
            {
                /* reset dth11 data */
                dth11_data[i] = 0;
                /* each byte has 8 bits => 5 * 8 = 40 bits */
                for (int j = 0; j < 8; ++j)
                {
                    /* get bit */
                    err = dth11_get_slave_data_bits(&data_bit);
                    if(SSP_SUCCESS == err)
                    {
                        dth11_data[i] |= (uint8_t)(data_bit<<(7-j));
                    }
                    else break;
                }
                if(err == SSP_ERR_ABORTED) break;
            }
        }
        /* transmission successful so far ? */
        if(err == SSP_SUCCESS)
        {
            err = dth11_check_and_asign_data( &humidity[0], &temperature[0]);
        }
        /* return status */
        return err;
    }
}

```

- The sync slave function uses the 86µ high level as a synchronization point. For this purpose the timer is used which counts the static variable `level_width_cnt` in the timer callback function up. The implementation is listed below.

```

void timer0_counter_callback(timer_callback_args_t * p_args){
    (void) p_args;
    ++level_width_cnt;
}

```

```

ssp_err_t dthll_sync_slave_start_signal(void)
{
    ssp_err_t err;

    err = SSP_SUCCESS;
    /*reset count */
    level_width_cnt = 0;
    /* start timer */
    g_timer0.p_api->start(g_timer0.p_ctrl);
    level_width_cnt = 0;
    /* wait for high level to sync on */
    while(P302_level == IOPORT_LEVEL_LOW )
    {
        g_ioport.p_api->pinRead(IOPORT_PORT_03_PIN_02,
                                &P302_level);
        /* level not rising timely - leave loop */
        if(level_width_cnt > 100)
        {
            err = SSP_ERR_ABORTED;
            break;
        }
    }

    level_width_cnt = 0;
    /* count high pulse width */
    while(P302_level == IOPORT_LEVEL_HIGH)
    {
        g_ioport.p_api->pinRead(IOPORT_PORT_03_PIN_02,
                                &P302_level);
        /* level not rising timely - leave loop */
        if(level_width_cnt > 100)
        {
            err = SSP_ERR_ABORTED;
            break;
        }
    }
    /* stop timer */
    g_timer0.p_api->stop(g_timer0.p_ctrl);
    /* when no errors occurred check for 80 us high width */
    if( err == SSP_SUCCESS)
    {
        if( level_width_cnt >= (86 - DELTA_T)
            && level_width_cnt <= (86 + DELTA_T) )
        {
            /* sync successful */
            err = SSP_SUCCESS;
        }
        else
        {
            /* unsuccessful slave start sync */
            err = SSP_ERR_ABORTED;
        }
    }
    /* return sync status */
    return err;
}

```

- After the synchronization the data bits follow. The get data bits function is designed to get the bits after successful synchronization. The code is shown below.

```

ssp_err_t dthll_get_slave_data_bits(uint8_t* const data_bit)
{
    ssp_err_t err;

    err = SSP_SUCCESS;

    /* start timer */
    g_timer0.p_api->start(g_timer0.p_ctrl);
    /* reset count */
    level_width_cnt = 0;
    /* wait till low level is over */
    while(P302_level == IOPORT_LEVEL_LOW )
    {
        g_ioport.p_api->pinRead(IOPORT_PORT_03_PIN_02,
                               &P302_level);
        /* level not rising timely - leave loop */
        if(level_width_cnt > 70)
        {
            err = SSP_ERR_ABORTED;
            break;
        }
    }

    level_width_cnt = 0;
    /* count high level */
    while(P302_level == IOPORT_LEVEL_HIGH )
    {
        g_ioport.p_api->pinRead(IOPORT_PORT_03_PIN_02,
                               &P302_level);
        /* level not falling timely - leave loop */
        if(level_width_cnt > 90)
        {
            err = SSP_ERR_ABORTED;
            break;
        }
    }
    /* stop timer */
    g_timer0.p_api->stop(g_timer0.p_ctrl);

    if( err == SSP_SUCCESS)
    {
        /* bit =? 0 */
        if( level_width_cnt >= (23 - DELTA_T)
           && level_width_cnt <= (23 + DELTA_T) )
        {
            *data_bit = 0;
        }
        /* bit =? 1 */
        else if( level_width_cnt >= (70 - DELTA_T)
                && level_width_cnt <= (70 + DELTA_T) )
        {
            *data_bit = 1;
        }
        else
        {
            /* unsuccessful abort data receive */
            err = SSP_ERR_ABORTED;
            /* error data */
            *data_bit = 0xFF;
        }
    }
}

```

- The last function is the check and assign function. The checksum is calculated by adding the first 4 bytes together. The sum should equal the fifth byte. If the checksums do not match then an error occurred during the transmission and data is not assigned.

```

ssp_err_t dth11_check_and_asign_data(uint8_t* const humidity,uint8_t* const temperature)
{
    ssp_err_t err;
    uint8_t protocol_checksum = dth11_data[4];
    uint8_t calculated_checksum = 0;

    err = SSP_SUCCESS;

    /* calculate checksum */
    calculated_checksum = (uint8_t) (dth11_data[0] + dth11_data[1] + dth11_data[2] + dth11_data[3])
    if( protocol_checksum == calculated_checksum)
    {
        /* assign data */
        humidity[0] = dth11_data[0];
        temperature[0] = dth11_data[2];
    }
    else
    {
        /* transmission error occurred */
        err = SSP_ERR_ABORTED;
    }
    return err;
}

```

- All the functions listed above are included in the files dth11_protocol_sm.h/c, which can be added to the project. Create a new folder by right clicking on the src-folder in the project explorer. Then add the two files.

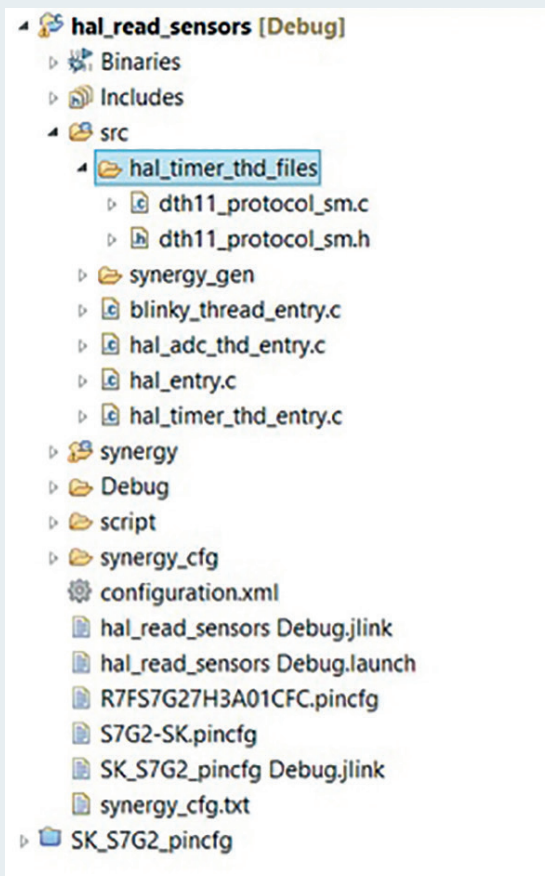


Fig. 142

- When the two protocol files are included correctly, then the following code should result in successful DTH11 sensor readings. This code in the hal_timer_thd_entry.c runs the dth11 sensor.

```

ssp_err_t dth11_protocol(uint8_t* const humidity, uint8_t* const temperature)
{
    ssp_err_t err;
    uint8_t data_bit;

    err = SSP_SUCCESS;

    /* master initiates transmission */
    err = dth11_master_start_signal();
    if(SSP_SUCCESS == err)
    {
        /* sync on slave start signal */
        err = dth11_sync_slave_start_signal();
        if(SSP_SUCCESS == err)
        {
            /* total of 5 bytes to receive */
            for (int i = 0; i < 5; ++i)
            {
                /* reset dth11 data */
                dth11_data[i] = 0;
                /* each byte has 8 bits => 5 * 8 = 40 bits */
                for (int j = 0; j < 8; ++j)
                {
                    /* get bit */
                    err = dth11_get_slave_data_bits(&data_bit);
                    if(SSP_SUCCESS == err)
                    {
                        dth11_data[i] |= (uint8_t)(data_bit<<(7-j));
                    }
                    else break;
                }
                if(err == SSP_ERR_ABORTED) break;
            }
        }
    }
    /* transmission successful so far ? */
    if(err == SSP_SUCCESS)
    {
        err = dth11_check_and_asign_data( &humidity[0], &temperature[0]);
    }
    /* return status */
    return err;
}

```

- The temperature and humidity values can be viewed in debugging mode. e²studio provides a real-time updating graph to visualize variables that change values at runtime. So both of these variables can also be added and observed in “Expressions” pane in Debug perspective. Or putting a breakpoint in the if-branch and one at the sleep-function allows to check if the DTH11 protocol has been successful. If the protocol has been not successful the if-branch gets skipped.

12.7. FRAMEWORK & FUNCTIONAL LIBRARIES

The Synergy Framework is an advanced software stack using the RTOS ThreadX, HAL-drivers or other X-Ware like USB Comms Framework to implement stacks that offer more services to the developer than normal HAL-drivers do. In this part we will read the ADC sensors with the use of the ADC framework instead of the ADC HAL driver to highlight certain differences between HAL and Framework software stacks.

STEPS TO DO:

- Create a new project using the template that has the pins already configured ("Import the template"). Create a new thread and call it `adc_framework_thd` and add the software stack Framework > Analog > ADC Periodic Framework

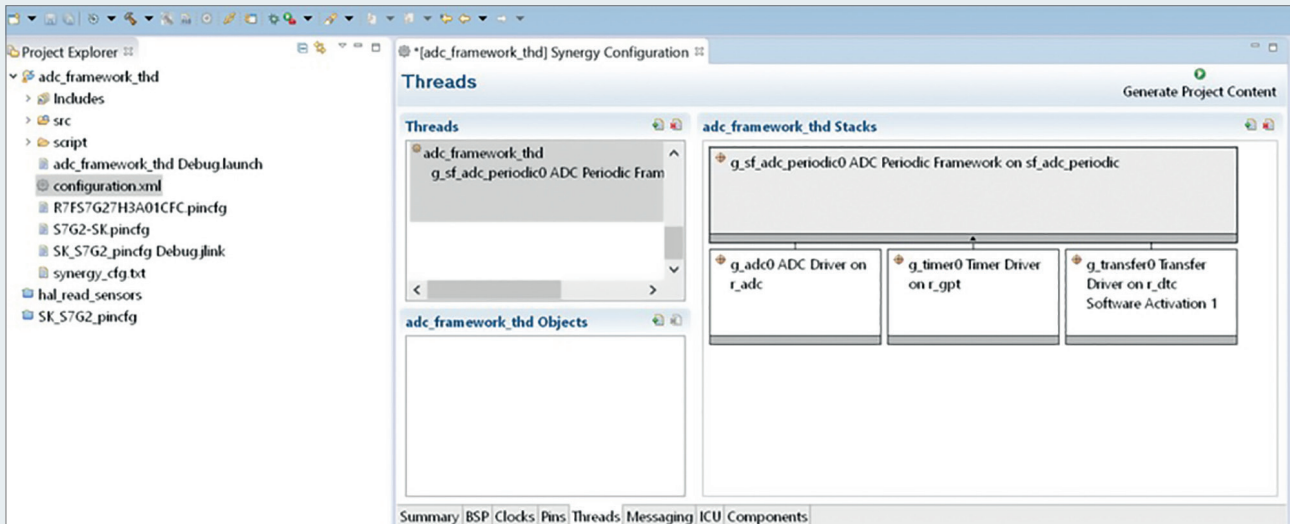


Fig. 143

- The software stacks consist of one or more than one module. It is typical for frameworks to use basic drivers to implement more advanced stacks that offer more services. In this case the ADC Periodic Framework stack uses three HAL stacks. The ADC, Timer and DTC HAL driver. The RTOS is used as well in framework stacks to coordinate interrupts and access to the used hardware modules. Keeping the basic structure in mind might help in understanding on how the framework stack works and what it does. Here the framework consists of 4 modules that can be configured.

- Configure the top stack:

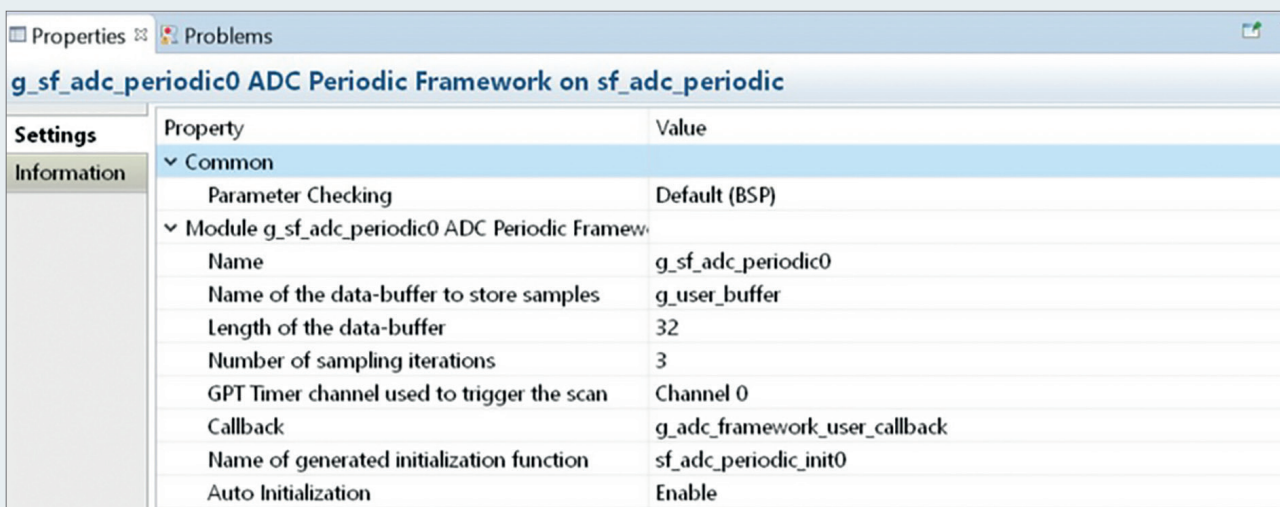
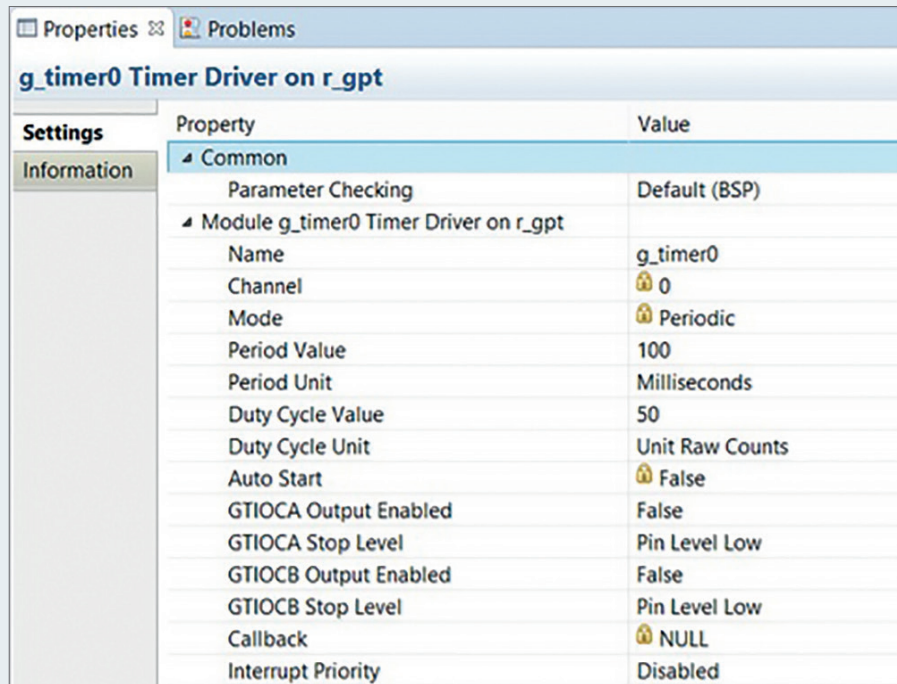


Fig. 144

- The framework utilizes a buffer and asks how many sampling iterations are required. (refer to configuration options). For reading the ADC sensor values a buffer of two and one sample iteration would be sufficient. For demonstration purposes we will use a buffer length of 32 and an iteration value of 3. This means that each active ADC channel will be scanned 3 times. The user gets the data by the callback function.

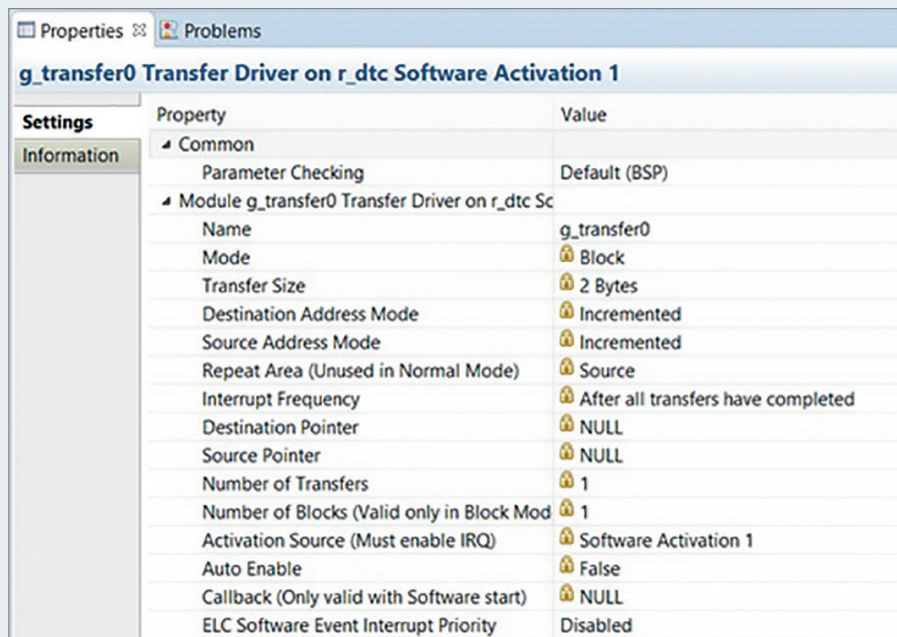
- Configure the ADC module like before, though this time we will enable add/average Count property with average two samples and enable channel 0 and 1. Also set the “Scan End Interrupt Priority” to higher than 15 to enable the interrupt which is the activation source for the data transfer into the framework buffer.
- Configure the GPT timer module. It is used to set the period of the ADC periodic framework. The example uses 100 ms.



Property	Value
Parameter Checking	Default (BSP)
Module g_timer0 Timer Driver on r_gpt	
Name	g_timer0
Channel	0
Mode	Periodic
Period Value	100
Period Unit	Milliseconds
Duty Cycle Value	50
Duty Cycle Unit	Unit Raw Counts
Auto Start	False
GTIOCA Output Enabled	False
GTIOCA Stop Level	Pin Level Low
GTIOCB Output Enabled	False
GTIOCB Stop Level	Pin Level Low
Callback	NULL
Interrupt Priority	Disabled

Fig. 145

- Configure the GPT timer module. It is used to set the period of the ADC periodic framework. The example uses 100 ms.



Property	Value
Parameter Checking	Default (BSP)
Module g_transfer0 Transfer Driver on r_dtc Sc	
Name	g_transfer0
Mode	Block
Transfer Size	2 Bytes
Destination Address Mode	Incremented
Source Address Mode	Incremented
Repeat Area (Unused in Normal Mode)	Source
Interrupt Frequency	After all transfers have completed
Destination Pointer	NULL
Source Pointer	NULL
Number of Transfers	1
Number of Blocks (Valid only in Block Mod	1
Activation Source (Must enable IRQ)	Software Activation 1
Auto Enable	False
Callback (Only valid with Software start)	NULL
ELC Software Event Interrupt Priority	Disabled

Fig. 146

- Generate the Synergy Configuration and open the generated file `adc_framework_thd_entry.c`.

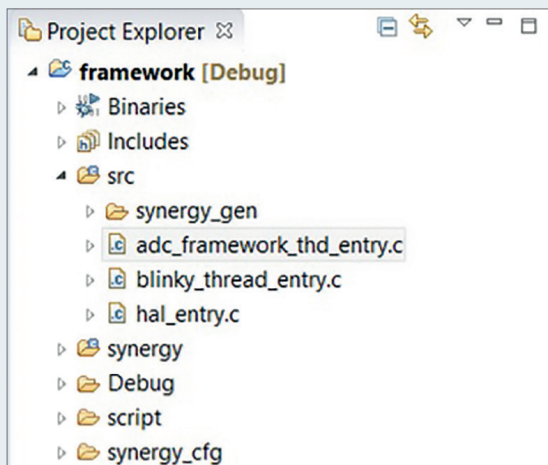


Fig. 147

- The code given below reads the ADC sensor values. There are two key parts. The first part is the need to start the framework. The second part is to get the data in the callback function. The `p_args` is a pointer to the structure that contains elements such as index of the current ADC values. The values are averaged again and then assigned to the corresponding variable. Notice that the uneven buffer values are assigned to the brightness and the even buffer values assigned to the water level. This is due to the way the framework works. In principle the framework runs several repeated scans, each starting at channel 0 (lowest enabled channel) and then increments through all channels up to the highest enabled channel (e.g. if channels 2, 4 and 7 are enabled, the scan results will include channels 2, 3, 4, 5, 6, 7). Notice that the use of the framework stack is easier than the ADC HAL stack.

```
#include "adc_framework_thd.h"

/*static variables */
static uint16_t adc_water_level;
static uint16_t adc_brightness;

/*callback function */
void g_adc_framework_user_callback(sf_adc_periodic_callback_args_t * p_args){
    uint32_t index = p_args->buffer_index;

    adc_brightness = (uint16_t)(g_user_buffer[index] + g_user_buffer[index+2] + g_user_buffer[index+4]) / 3;
    adc_water_level = (uint16_t)(g_user_buffer[index+1] + g_user_buffer[index+3] + g_user_buffer[index+5]) / 3;
}

/*adc_framework_thd entry function */
void adc_framework_thd_entry(void)
{
    /* variables */
    ssp_err_t err;

    /*start the scan, since auto start is off */
    err = g_sf_adc_periodic0.p_api->start(g_sf_adc_periodic0.p_ctrl);
    if( err != SSP_SUCCESS) for(;;);

    while (1)
    {
        /*wait for 200 ms */
        tx_thread_sleep(20);
    }
}
```

- Build and debug the code, observe the ADC values in the expression window in the debug window section. In debug perspective click on "Expressions" in the top right tab and add the ADC values `adc_water_level` and `adc_brightness` as new expression. Set a breakpoint to `tx_thread_sleep(2)`; in `adc_framework_thd_entry.c` and run the program.

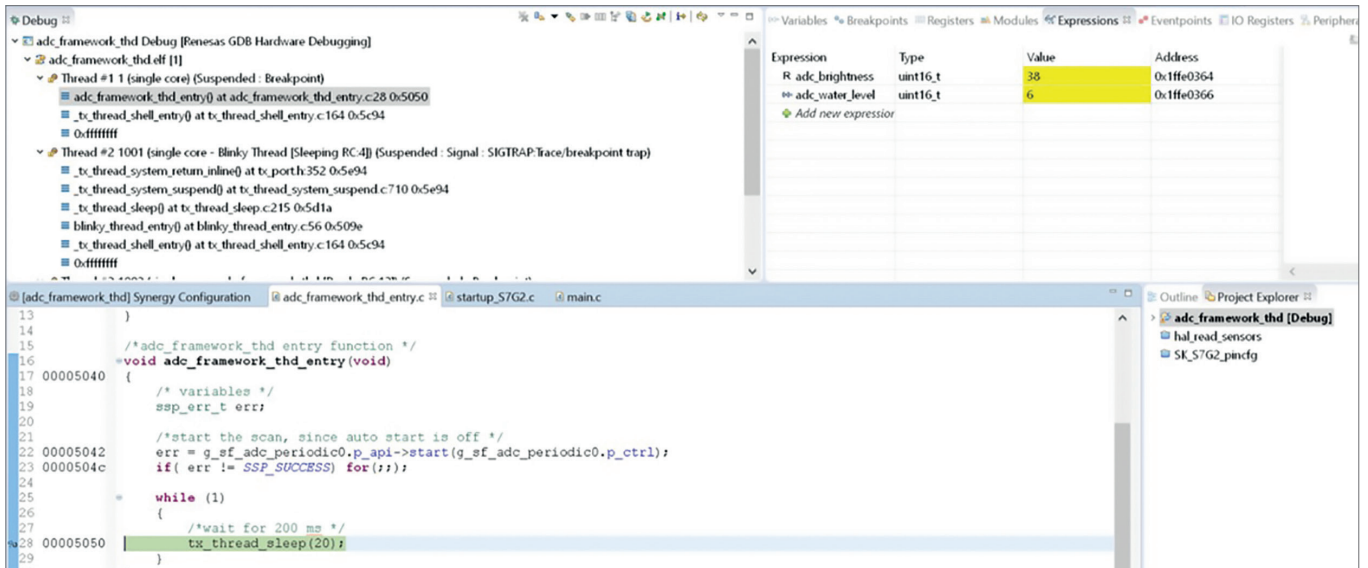


Fig. 148

12.8. CONNECTIVITY

A console is a very useful tool to setup a communication between your PC and the microcontroller. In this step we want to implement a console using an Ethernet connection as physical link. We will read the sensor data from the microcontroller to the PC using the telnet tool on the PC.

STEPS TO DO:

- Create a new thread called `console_thd`

New Thread		
Settings	Property	Value
	Thread	
	Symbol	<code>console_thd</code>
	Name	<code>console_thd</code>
	Stack size (bytes)	1024
	Priority	1
	Auto start	Enabled
	Time slicing interval (ticks)	1

Fig. 149

- Add the console software stack under `framework > Services > Console Framework`.
- Add the Ethernet Communication Framework `sf_el_nx_comms` shown in the illustration below. The console will use a telnet server in order to establish connection. Note that telnet connection is not encrypted. Later on we will need a telnet client in order to set up a connection.

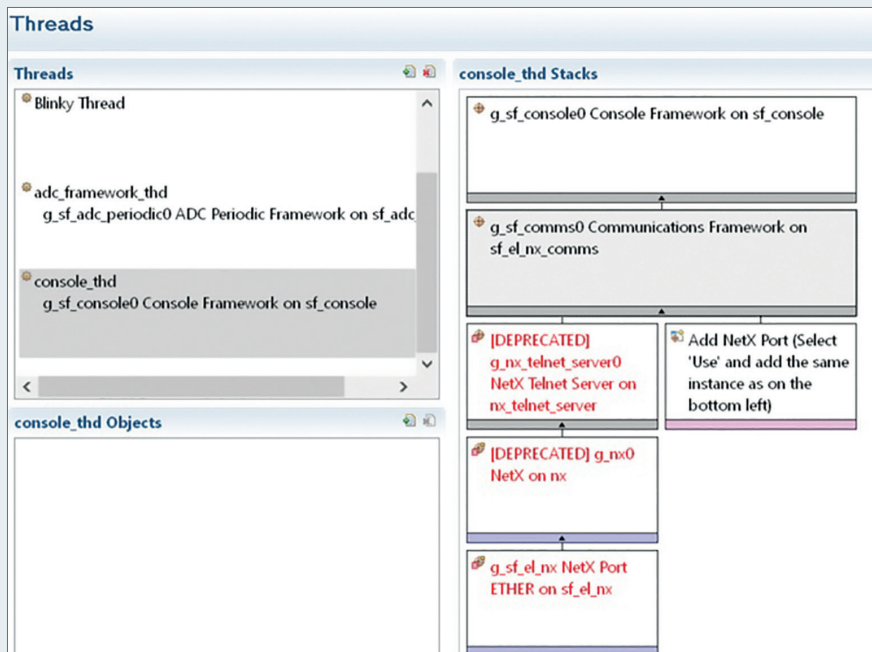


Fig. 150

- Configure the software stack modules. The Console Framework configuration can stay as it is.

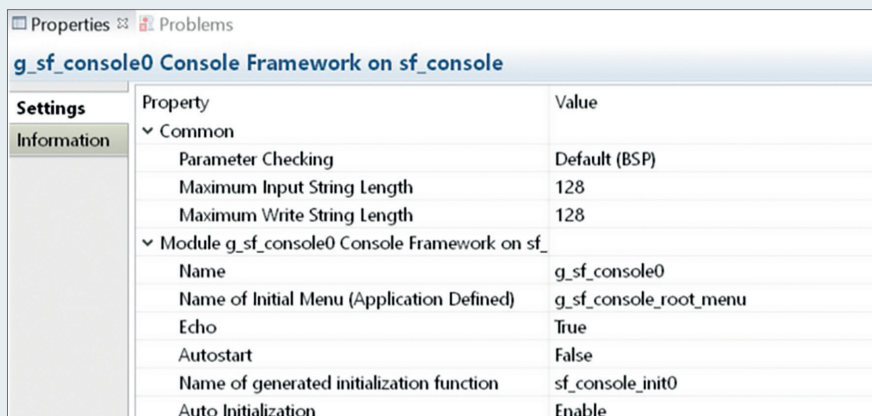


Fig. 151

- In the Communications Framework configuration set the IP host-address you like to have. In this case the host address is set to 192.168.0.10. Change the Channel to 1 and leave the rest as it is.

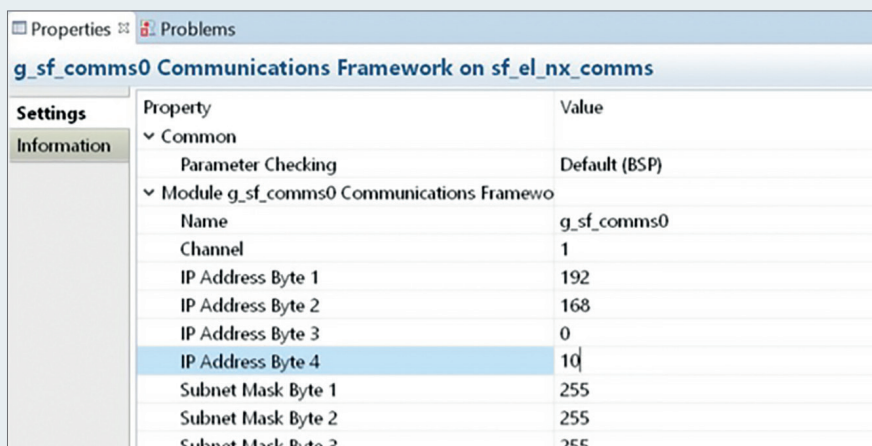


Fig. 152

- In the NetX Port ETHER property window change Channel 1 Phy Reset Pin to IOPORT_PORT_08_PIN_06 and change the Port Channel to 1 with an Ethernet Priority higher than 15.
- When using multiple boards on the same network, it is recommended to change the least significant “low bits” in the Channel 1 MAC address.

Property	Value
Common	
Parameter Checking	Default (BSP)
Channel 0 Phy Reset Pin	IOPORT_PORT_09_PIN_03
Channel 0 MAC Address High Bits	0x00002E09
Channel 0 MAC Address Low Bits	0x0A0076C7
Channel 1 Phy Reset Pin	IOPORT_PORT_08_PIN_06
Channel 1 MAC Address High Bits	0x00002E09
Channel 1 MAC Address Low Bits	0x0A0076C8
Number of Receive Buffer Descriptors	8
Number of Transmit Buffer Descriptors	32
Ethernet Interrupt Priority	Priority 0 (highest)
Module g_sf_el_nx NetX Port ETHER on sf_el_nx	
Name	g_sf_el_nx
Channel	1
Callback	NULL

Fig. 153

- Note if there are any deprecated module blocks then go to their property module and disable the warning. The configuration set up is finished. Generate the synergy configuration.
- Create a new folder in the src folder in the project explorer. Create the files console_def.c/h and uint_to_string_conversion.c/h. The first source file implements the console behaviour. The second source file implements the conversion from uint data types to a string.
- The console_def.h contains a single function that serves as an entry point to the console function.

```
* console_def.h

#ifndef CONSOLE_THD_FILES_CONSOLE_DEF_H_
#define CONSOLE_THD_FILES_CONSOLE_DEF_H_

/* includes */
#include "console_thd.h"

/* console thread main function */
void console_main(void);

#endif /* CONSOLE_THD_FILES_CONSOLE_DEF_H_ */
```

- The corresponding function implementation is as followed

```
void console_main(void)
{
    g_sf_console0.p_api->prompt(g_sf_console0.p_ctrl, NULL, TX_WAIT_FOREVER);
}
```

- The function calls the prompt and waits for user input. Right now the console has no behaviour implemented, which follows in the next step.
- The console starts in the g_sf_console_root_menu as set in the stack configuration.

```
/* console menus */
const sf_console_menu_t g_sf_console_root_menu = {
    .menu_prev = NULL,
    .menu_name = (uint8_t*)"root: ",
    .num_commands = (sizeof(g_sf_root_commands)) / (sizeof(g_sf_root_commands[0])),
    .command_list = &g_sf_root_commands[0]
};
```

- The menu has 4 entities: pointer to the previous menu, the menu name to prompt out, the number of menu commands and lastly the command list of the menu. Since this is the root menu there are no previous menus and the value is set to 'NULL'. The shown prompt is 'root: ' and the first entry to our command list is g_sf_root_commands[0].

```
/* console root command */
const sf_console_command_t g_sf_root_commands[]=
{
    { .command = (uint8_t*)"login", .help = (uint8_t*)"log in sk-s7g2",
      .callback = login_callback, .context = NULL},
};
```

- The first command that should be implemented is a login function, that allows the access of another menu with more functionality. When the user executes the 'login' command the callback function login_callback is called. The function contains two main parts. The first is to access the user-ID and the second is to verify password. After a successful login the console menu should change to a menu with more functionality.

```
const sf_console_menu_t g_sf_console_sk_s7g2_menu ={
    .menu_prev = &g_sf_console_root_menu,
    .menu_name = (uint8_t*)"sk-s7g2: ",
    .num_commands = (sizeof(g_sf_sk_s7g2_commands)) / (sizeof(g_sf_sk_s7g2_commands[0])),
    .command_list = &g_sf_sk_s7g2_commands[0]
};
```

- The command list consists of a simple function that should read out the sensor values to the console.

```
const sf_console_command_t g_sf_sk_s7g2_commands[]=
{
    { .command = (uint8_t*)"rsv", .help = (uint8_t*)"read sensor values",
      .callback = read_sensor_values_callback, .context = NULL},
};
```

- The first function to implement is the login function. The code for the function is given below.
- The login function is implemented with the help of the write and read API functions. In the first part the string 'user-id:' is printed to the console, then the console waits for input from the user. If no input is given within 10 seconds then the function will timeout (the code will also time out if insufficient number of characters has been collected). In case the wrong ID is written then the string 'wrong user-id' is written. The same procedure is done with the password. After a successful login the current_menu pointer is changed to the next menu.

```

/* login callback functions */
void login_callback(sf_console_callback_args_t * p_args)
{
    (void) p_args;
    ssp_err_t err;

    /* cast from void to sf_console_instance_ctrl_t needed for access */
    sf_console_instance_ctrl_t * casted_console_p_ctrl = (sf_console_instance_ctrl_t *) g_sf_console0.p_ctrl;

    /* check for user id */
    g_sf_console0.p_api->write(g_sf_console0.p_ctrl, str_id, TX_NO_WAIT);
    /* wait 10 seconds for inputs */
    err = g_sf_console0.p_api->read(g_sf_console0.p_ctrl, str_data, sizeof(str_data), 1000);
    if( err == SSP_ERR_INTERNAL)
    {
        g_sf_console0.p_api->write(g_sf_console0.p_ctrl, str_timeout, TX_NO_WAIT);
    }
    else if(err == SSP_SUCCESS)
    {
        /* check user id */

        /* set termination so that string compare does not run to end */
        str_data[sizeof(str_renesas) / sizeof(str_renesas[0])] = '\0';
        if(strcmp((char*)str_data, (char*)str_renesas) != STRING_EQUAL)
        {
            g_sf_console0.p_api->write(g_sf_console0.p_ctrl, str_wrong_user_id, TX_NO_WAIT);
        }
        else
        {
            /* user id correct, now check password */
            g_sf_console0.p_api->write(g_sf_console0.p_ctrl, str_password, TX_NO_WAIT);
            err = g_sf_console0.p_api->read(g_sf_console0.p_ctrl, str_data, sizeof(str_data), 1000);
            if( err == SSP_ERR_INTERNAL)
            {
                g_sf_console0.p_api->write(g_sf_console0.p_ctrl, str_timeout, TX_NO_WAIT);
            }
            else if(err == SSP_SUCCESS)
            {
                /* set termination so that string compare does not run to end */
                str_data[sizeof(str_synergy) / sizeof(str_synergy[0])] = '\0';
                if(strcmp((char*)str_data, (char*)str_synergy) == STRING_EQUAL)
                {
                    /* user id and password correct change menu */
                    casted_console_p_ctrl->p_current_menu = &g_sf_console_sk_s7g2_menu;
                }
                else
                {
                    g_sf_console0.p_api->write(g_sf_console0.p_ctrl, str_wrong_password, TX_NO_WAIT);
                }
            }
        }
    }
}
}
}

```

- The next read sensor values function will need a conversion function to string in order to print the numeric values out correctly. The corresponding function is depicted below.

```

void uint_2_str(uint16_t val, char *buffer)
{
    char const digit[] = "0123456789";
    char* p = buffer;
    uint16_t shift_cnt = val;

    /* count length of string */
    do
    {
        p++;
        shift_cnt = shift_cnt / 10;
    }while(shift_cnt);

    /* terminate */
    *p = '\0';

    /* calculate numeric string */
    do
    {
        *--p = digit[val % 10];
        val = val / 10;
    }while(val);
}

```

- The function calculates in the first step how many char slots are necessary to print out the value correctly. In the next step the char slots are filled with the corresponding char values. Now we can implement the read sensor function which is illustrated below.

```

void read_sensor_values_callback(sf_console_callback_args_t * p_args)
{
    (void) p_args;
    /* write zeros into */
    memset((void*) str_data, 0, sizeof(str_data));
    /* append brightness string */
    strcat((char*)str_data, (char*)str_temperature);
    /* convert value to string and append */
    uint_2_str((uint16_t)dth11_temperature, (char*) str_numeric_conversion);
    strcat((char*)str_data, (char*)str_numeric_conversion);
    strcat((char*)str_data, (char*)str_celsius);
    /* add space string */
    strcat((char*)str_data, (char*)str_space);
    /* append water level string */
    strcat((char*)str_data, (char*)str_humidity);
    uint_2_str((uint16_t)dth11_humidity, (char*) str_numeric_conversion);
    strcat((char*)str_data, (char*)str_numeric_conversion);
    strcat((char*)str_data, (char*)str_rel_humidity);
    strcat((char*)str_data, (char*)str_enter);
    /* print temperature and humidity */
    g_sf_console0.p_api->write(g_sf_console0.p_ctrl, str_data, TX_NO_WAIT);

    /* write zeros into */
    memset((void*) str_data, 0, sizeof(str_data));
    /* append brightness string */
    strcat((char*)str_data, (char*)str_brightness);
    /* convert value to string and append */
    uint_2_str(adc_brightness, (char*) str_numeric_conversion);
    /* add space string */
    strcat((char*)str_data, (char*)str_space);
    strcat((char*)str_data, (char*)str_numeric_conversion);
    /* add space string */
    strcat((char*)str_data, (char*)str_space);
    strcat((char*)str_data, (char*)str_space);
    strcat((char*)str_data, (char*)str_space);
    /* append water level string */
    strcat((char*)str_data, (char*)str_water_level);
    uint_2_str(adc_water_level, (char*) str_numeric_conversion);
    strcat((char*)str_data, (char*)str_numeric_conversion);
    strcat((char*)str_data, (char*)str_enter);
    /* print brightness and water level */
    g_sf_console0.p_api->write(g_sf_console0.p_ctrl, str_data, TX_NO_WAIT);
}

```

- First it sets a buffer char str_data[128] values to zero then it builds the string to print out by appending the needed strings step by step. At the end the string is printed out.
- Note that currently the console is accessing the sensor values directly. The console reads sensor values over global variables listed below.

```

/* global variables */
extern uint16_t adc_water_level;
extern uint16_t adc_brightness;
extern uint8_t dth11_humidity;
extern uint8_t dth11_temperature;

```

- The last step is to adjust the adc_framework and add dth11 sensor values. Therefore create a new folder and call it adc_framework_thd_ files and add the dth11_protocol_sm.c/h files, that we created earlier on in this project.

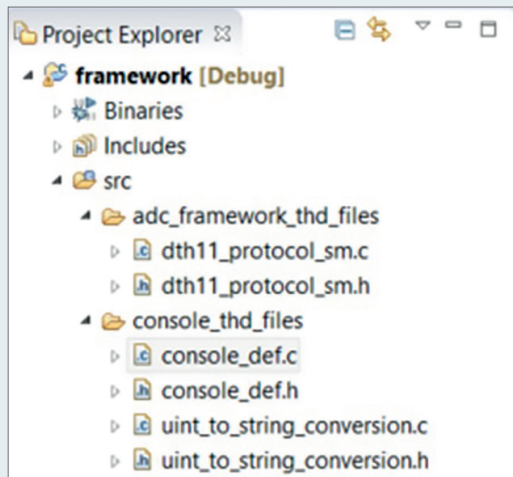


Fig. 154

- In the `dth11_protocol_sm.h` change the include to `adc_framework_thd.h`.

```
#ifndef ADC_FRAMEWORK_THD_FILES_DTH11_PROTOCOL_SM_H_
#define ADC_FRAMEWORK_THD_FILES_DTH11_PROTOCOL_SM_H_

#include "adc_framework_thd.h"

#define DELTA_T 5

ssp_err_t dth11_protocol_init(void);
ssp_err_t dth11_protocol(uint8_t* const humidity, uint8_t* const temperature);
```

- Now open the synergy configurator and add a `agt` driver to the `adc_framework_thd` stacks. Then change the names of the timer driver as illustrated below.

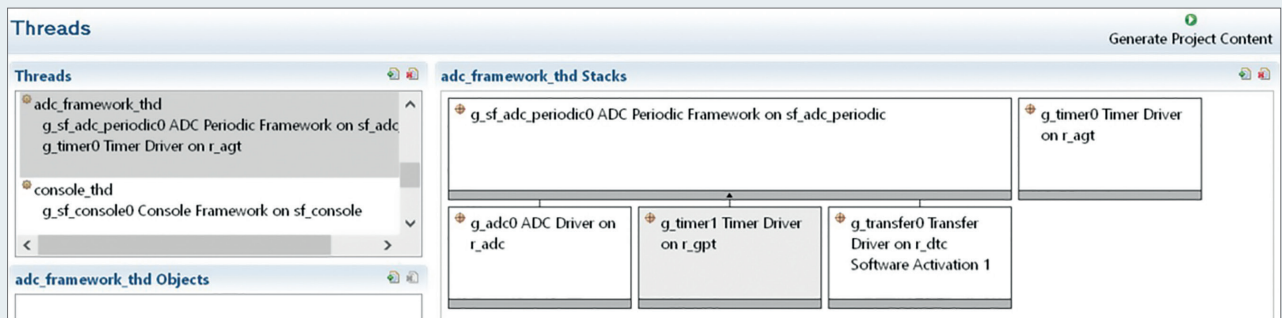


Fig. 155

- Configure the `g_timer0` as shown.

g_timer0 Timer Driver on r_agt		
Settings	Property	Value
Information	▼ Common	
	Parameter Checking	Default (BSP)
	▼ Module g_timer0 Timer Driver on r_agt	
	Name	g_timer0
	Channel	1
	Mode	Periodic
	Period Value	1
	Period Unit	Microseconds
	Auto Start	True
	Count Source	PCLKB
	AGTO Output Enabled	False
	AGTIO Output Enabled	False
	Output Inverted	False
	Callback	timer0_counter_callback
Interrupt Priority	Priority 0 (highest)	

Fig. 156

- Generate the synergy configuration and open the `adc_framework_thd_entry.c` and adjust the file as followed. The thread now runs the `dth11` sensors as well and calls the `dth11` protocol after each elapsed second.

```
#include "adc_framework_thd.h"
#include "adc_framework_thd_files/dth11_protocol_sm.h"

/* global variables */
uint16_t adc_water_level;
uint16_t adc_brightness;
uint8_t dth11_humidity;
uint8_t dth11_temperature;

/* callback function */
void g_adc_framework_user_callback(sf_adc_periodic_callback_args_t * p_args){

    uint32_t index = p_args->buffer_index;

    adc_brightness = (uint16_t)(g_user_buffer[index] + g_user_buffer[index+2] + g_user_buffer[index+4]) / 3;
    adc_water_level = (uint16_t)(g_user_buffer[index+1] + g_user_buffer[index+3] + g_user_buffer[index+5]) / 3;
}

/* adc_framework_thd entry function */
void adc_framework_thd_entry(void)
{
    /* variables */
    ssp_err_t err;
    uint8_t cnt = 0;

    /* start the scan, since auto start is off */
    err = g_sf_adc_periodic0.p_api->start(g_sf_adc_periodic0.p_ctrl);
    if( err != SSP_SUCCESS) for(;;);

    err = dth11_protocol_init();
    if( err != SSP_SUCCESS) for(;;);

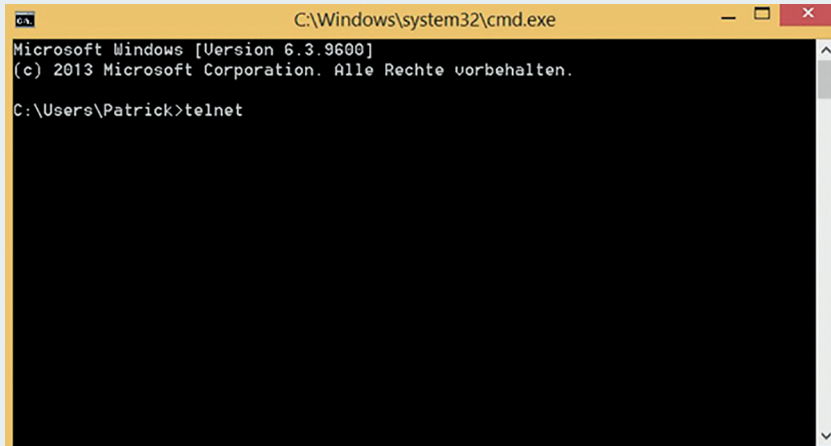
    while (1)
    {
        /* update sensor values */
        if(cnt == 0)
        {
            err = dth11_protocol(&dth11_humidity, &dth11_temperature);
            if( SSP_SUCCESS == err)
            {
                /* error handling
            }
            /* wait 1 second to get dth11 data */
            cnt = 6;
        }
        --cnt;

        /* wait for 200 ms */
        tx_thread_sleep(20);
    }
}
```

- Here the necessary code adjustments are finished and we can now continue to set up a telnet client.

STEPS TO DO FOR SETTING UP A TELNET CLIENT:

- Every windows systems contains a telnet client that can be activated. There is also the possibility to use console programs that support telnet like PuTTY. In this case we use the windows client since no external program is needed. The setup is shown in the following link: <https://support.microsoft.com/de-de/help/2801292>
- Set the IP configuration of the port you use for the Ethernet connection to the Starter Kit to 192.168.0.11 and 255.255.255.0 (this IP address has to fit to the IP address you defined for the Starter Kit before)
- Connect your hardware with your PC via Ethernet cable. Open up the command console then input telnet.



```

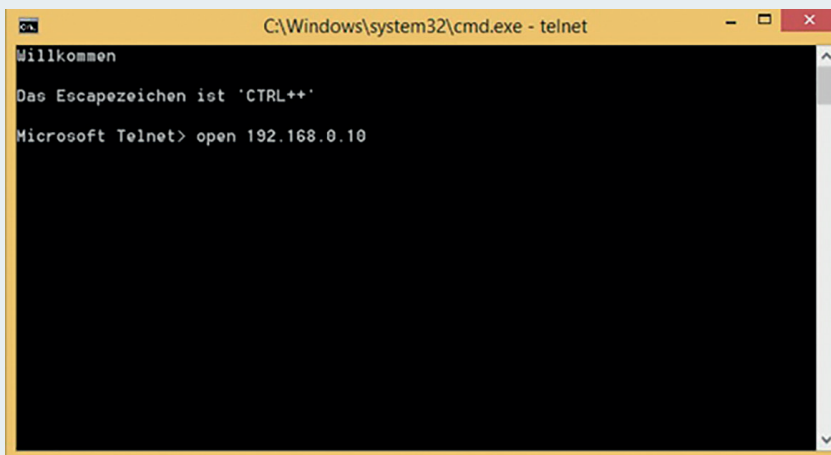
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. Alle Rechte vorbehalten.

C:\Users\Patrick>telnet

```

Fig.157

- Enter open 192.168.0.10 and enter again to get into the SK-S7G2 console. Make sure that the hardware is loaded with the program and is running at this point. Otherwise connection cannot be established. In order to get help from the console input '?'.



```

C:\Windows\system32\cmd.exe - telnet
Willkommen
Das Escapezeichen ist 'CTRL++'
Microsoft Telnet> open 192.168.0.10

```

Fig.158

- The implemented console looks like this.



```

Telnet 192.168.0.10
root: >login
user-id: renesas
password: synergy
sk-s7g2: >rsv
temperature: 20xC humidity: 43%RH
brightness: 76 water_level: 5
sk-s7g2: >

```

Fig.159

- Login to the Starter Kit using the login name 'renesas' and the password 'synergy'.
- Read the sensor data by using the rsv command.

12.9. GUI

In this chapter we are going to use GUIX™ to implement a graphical user interface that displays the sensor values on the LCD. To avoid writing the GUI code manually we will use the program GUIX Studio™, which can be downloaded from the synergy gallery under the tab “Development Tools”. Using GUIX Studio™ you can design your GUI graphically and generate the code automatically. This code can be included and used with the GUIX™ software stack.

This part of the project is by far the most complex and requires many steps in order to get it running and to understand what is happening within GUIX Studio™. Starting with GUIX™ can be very intimidating, therefore this project starts with a basic template that the user can slowly comprehend and change in order to get used to GUIX™.

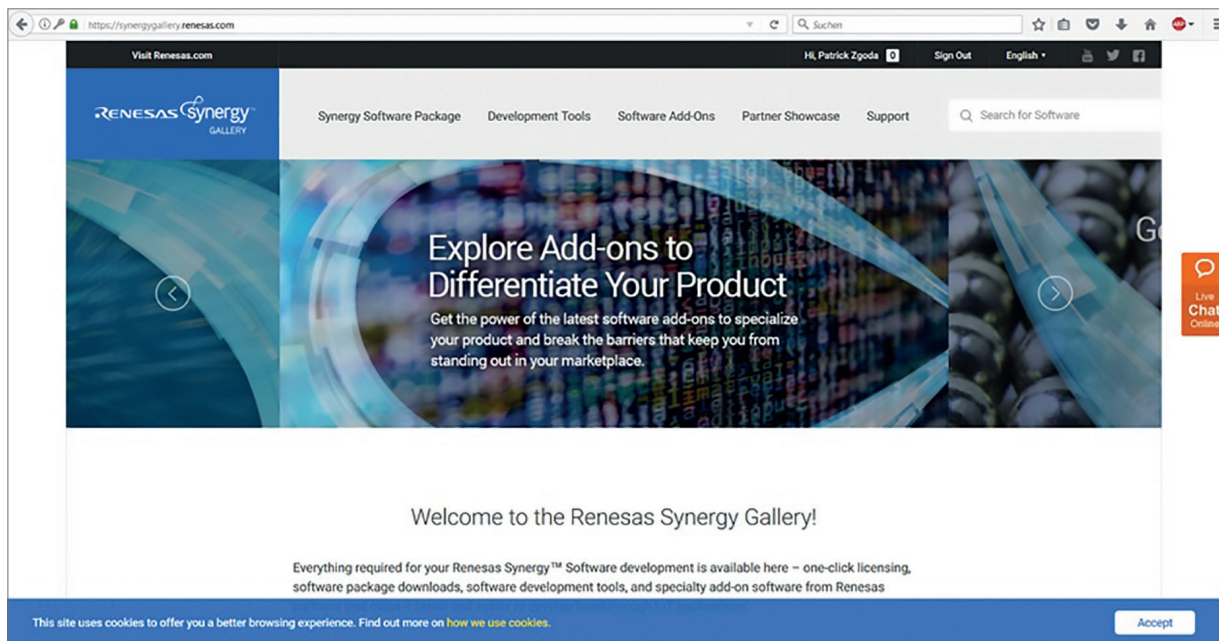


Fig. 160

STEPS TO DO:

- After installing GUIX Studio™ start the program. The following window will open.

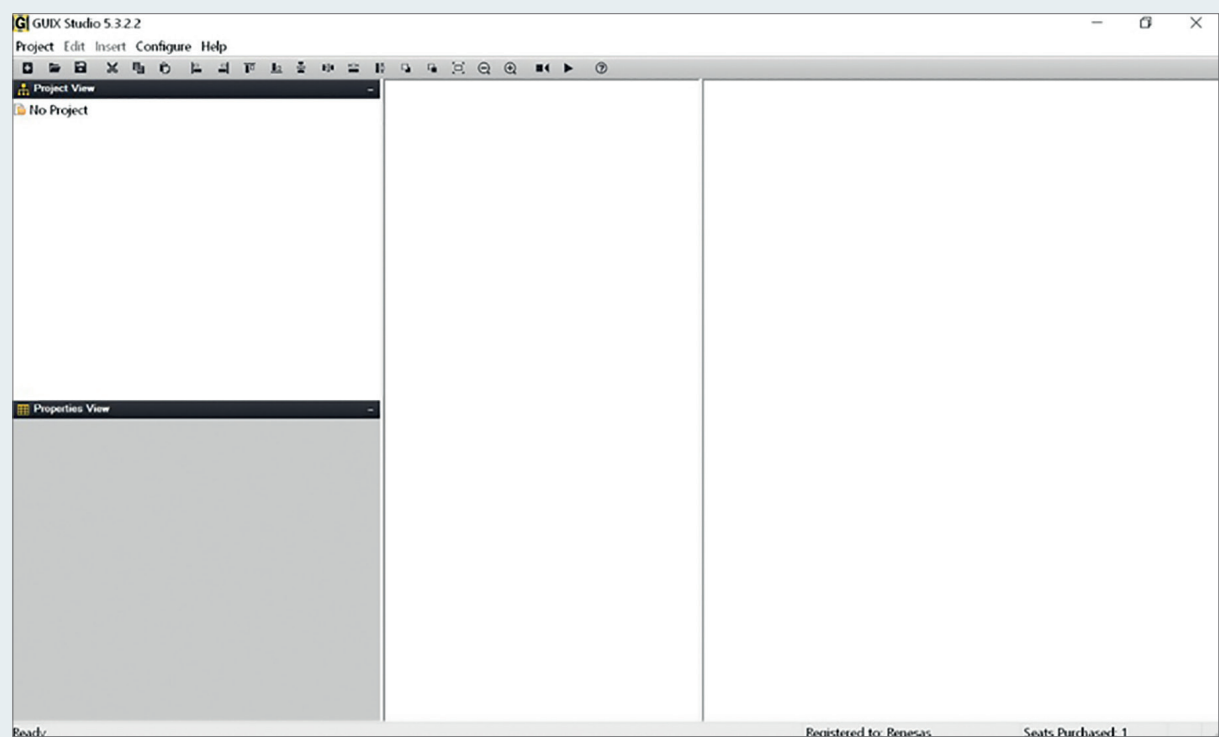


Fig. 161

- In the project folder open the `gui_thd_files`. This folder contains all the necessary basic code to get a LCD application running using the SK_S7G2 Starter Kit. Open the `rcm_gui` folder (room condition monitor) and open up the GUIX™ project by double clicking on `room_condition_monitor.gxp`.

Name	Änderungsdatum	Typ	Größe
gui	26.06.2017 17:13	Dateiordner	
gui_raw	26.06.2017 17:13	Dateiordner	
room_condition_monitor	19.04.2017 15:34	GUIX Studio Project	138 KB

Fig. 162

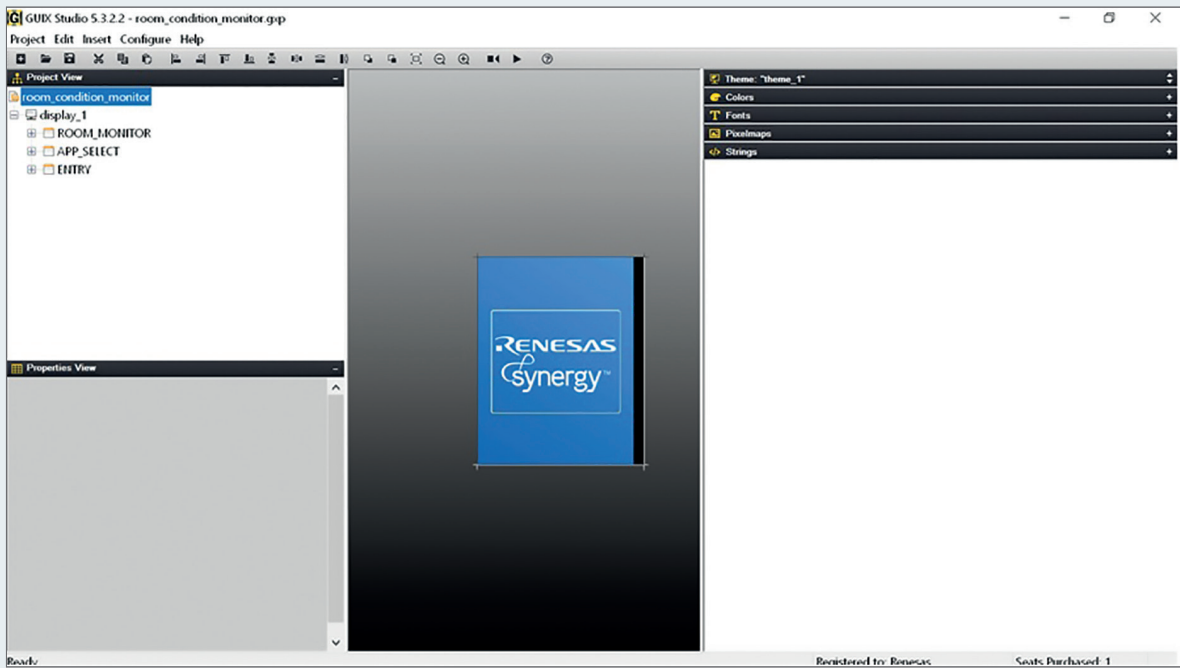


Fig. 163

- The following window opens. Navigating through the project view shows that there is one display with three windows. The windows are the ENTRY, APP_SELECT and the ROOM_MONITOR window.
- The ENTRY window only contains an icon button called B_ENTRY, which should navigate to the next window when the user pushes the button. The next window after pushing the button is an app select window, where the user can choose which app to open.

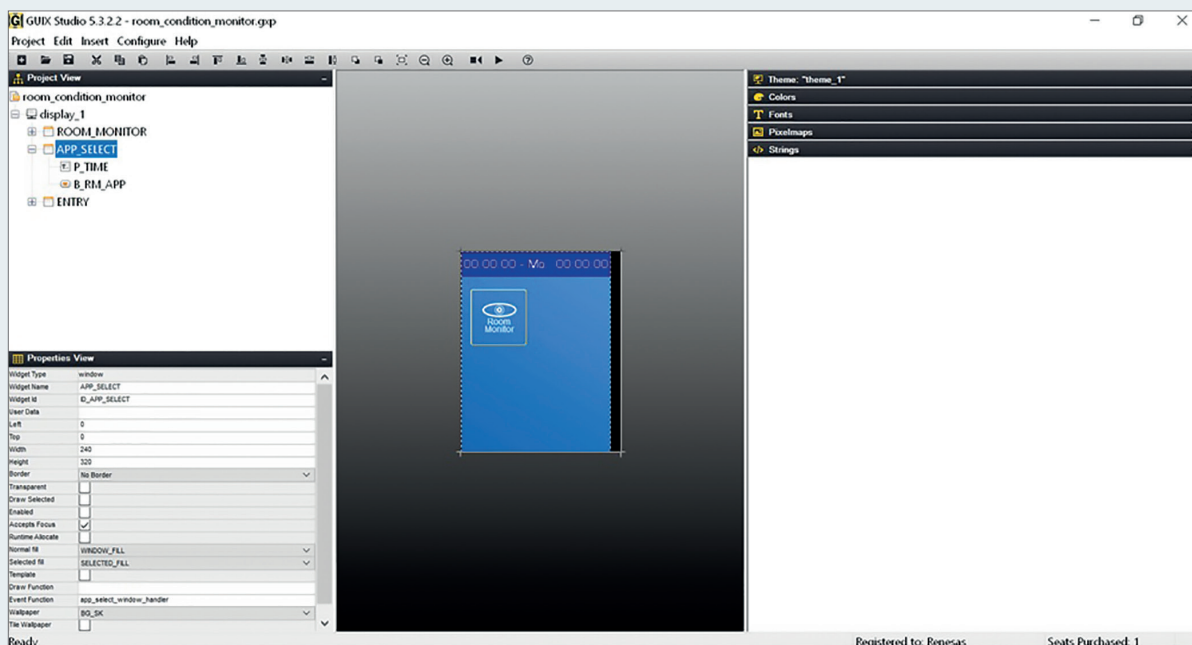


Fig. 164

- At the moment the APP_SELECT window only contains one app (the date and clock function is not implemented). The app is our room condition monitor app. By clicking on the icon button the window should change to the ROOM_MONITOR window.

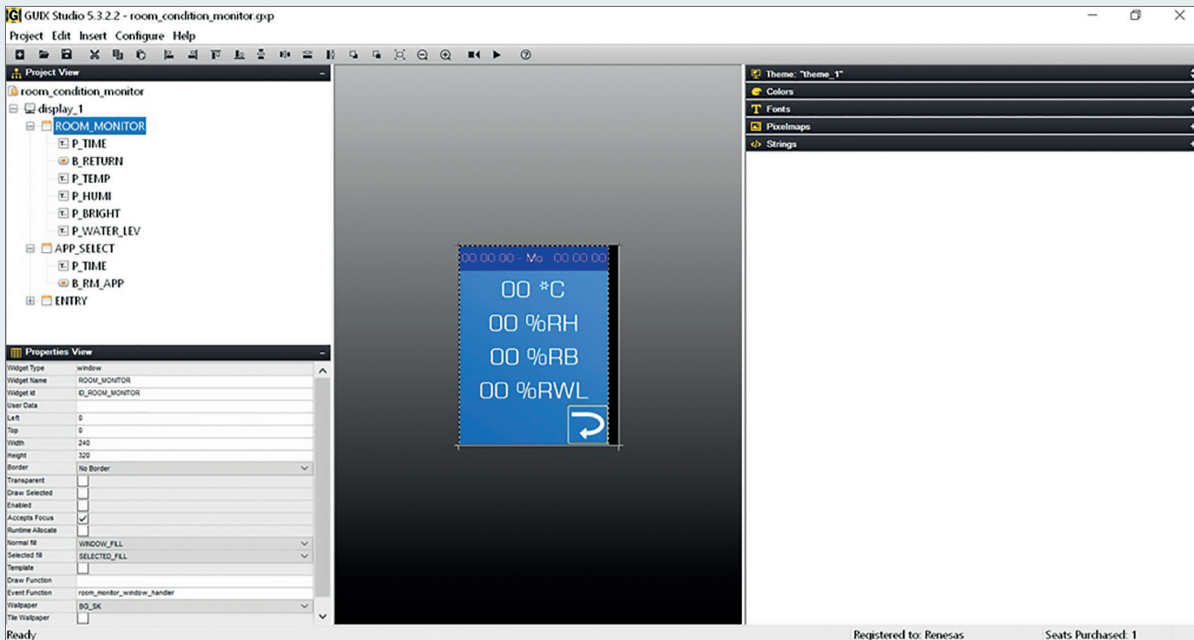


Fig. 165

- The window displays the temperature, humidity, brightness and water_level values, that we get from the sensors. The icon button at the bottom right corner serves a return button to navigate back to the APP_SELECT window.
- The pictures for the GUI are saved in the file rcm_gui > gui_raw. The generated code is kept in rcm_gui > gui.
- This guide will not go into further details to explain how to use GUIX Studio™ as there are already good GUIX™ documentations. We encourage the reader to experiment with this template and make changes in order to see how GUIX Studio™ works.
- Return to e²studio
- Import the framework zip file and rename the project to guix
- Create a new thread called gui_thd. Add the software stack Framework > Input > Touch Panel Framework. Set the names as shown.

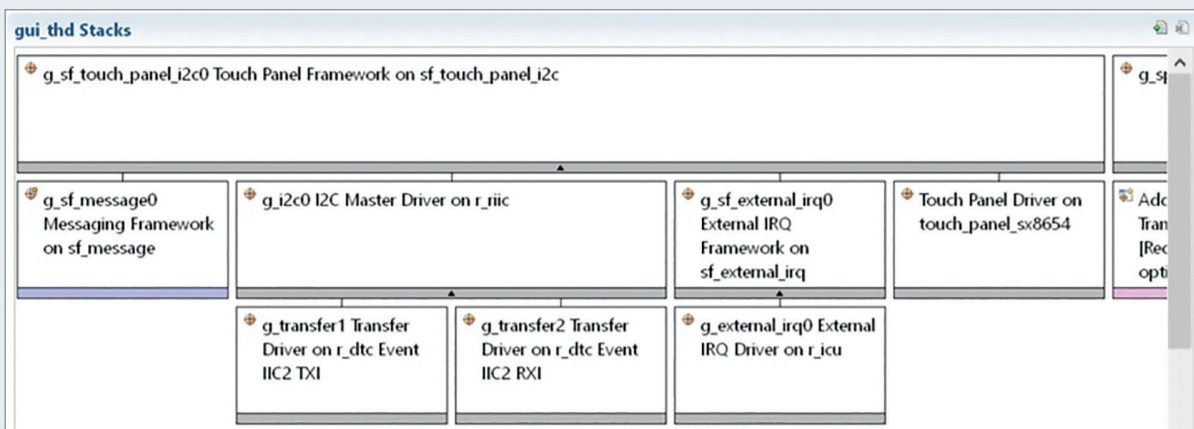
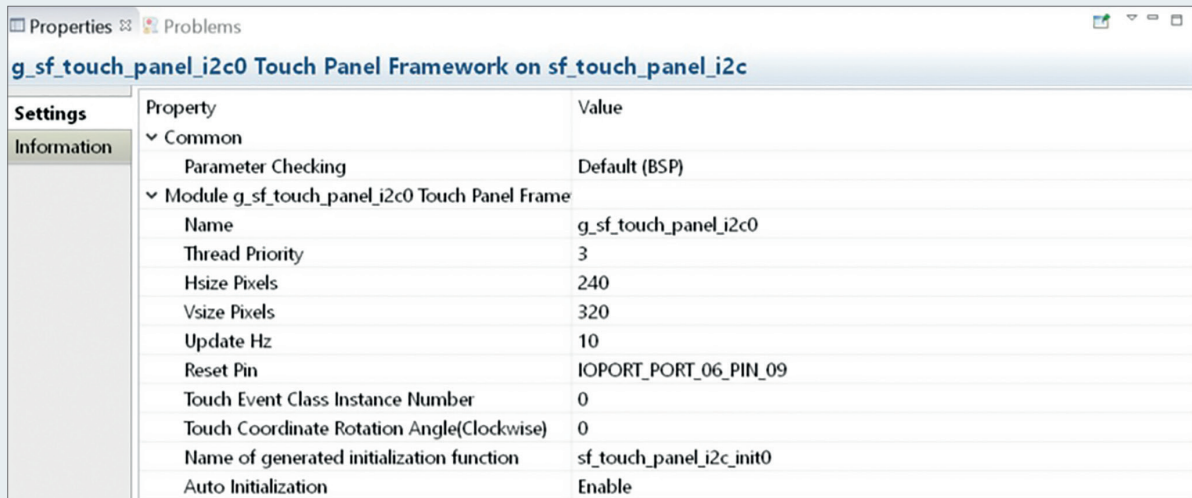


Fig. 166

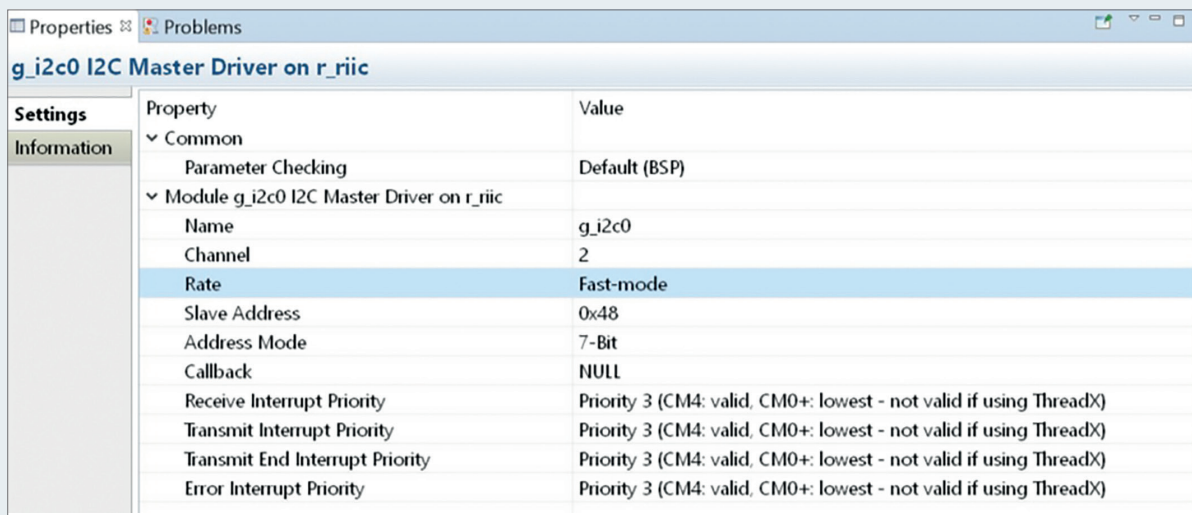
- Configure the Touch Panel Framework as illustrated below.



Property	Value
Parameter Checking	Default (BSP)
Module g_sf_touch_panel_i2c0 Touch Panel Frame	
Name	g_sf_touch_panel_i2c0
Thread Priority	3
Hsize Pixels	240
Vsize Pixels	320
Update Hz	10
Reset Pin	IOPORT_PORT_06_PIN_09
Touch Event Class Instance Number	0
Touch Coordinate Rotation Angle(Clockwise)	0
Name of generated initialization function	sf_touch_panel_i2c_init0
Auto Initialization	Enable

Fig. 167

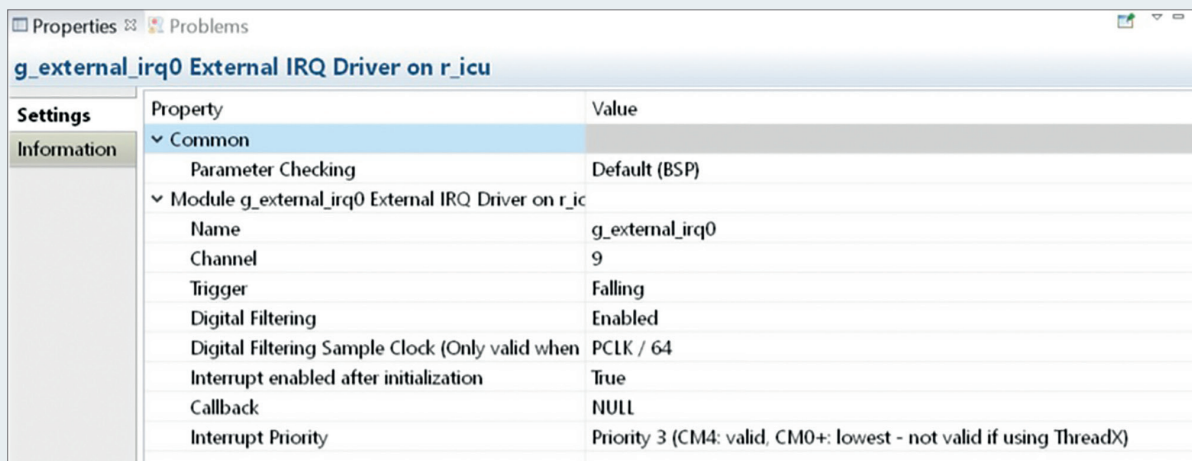
- Add Master Driver on r_riic and configure it as follows.



Property	Value
Parameter Checking	Default (BSP)
Module g_i2c0 I2C Master Driver on r_riic	
Name	g_i2c0
Channel	2
Rate	Fast-mode
Slave Address	0x48
Address Mode	7-Bit
Callback	NULL
Receive Interrupt Priority	Priority 3 (CM4: valid, CM0+: lowest - not valid if using ThreadX)
Transmit Interrupt Priority	Priority 3 (CM4: valid, CM0+: lowest - not valid if using ThreadX)
Transmit End Interrupt Priority	Priority 3 (CM4: valid, CM0+: lowest - not valid if using ThreadX)
Error Interrupt Priority	Priority 3 (CM4: valid, CM0+: lowest - not valid if using ThreadX)

Fig. 168

- Then configure g_external_irq0 External IRQ Driver.



Property	Value
Parameter Checking	Default (BSP)
Module g_external_irq0 External IRQ Driver on r_icu	
Name	g_external_irq0
Channel	9
Trigger	Falling
Digital Filtering	Enabled
Digital Filtering Sample Clock (Only valid when	PCLK / 64
Interrupt enabled after initialization	True
Callback	NULL
Interrupt Priority	Priority 3 (CM4: valid, CM0+: lowest - not valid if using ThreadX)

Fig. 169

- Finally add the Touch Panel Driver on touch_panel_sx8654.
- Add the software stack X-Ware > GUIX > GUIX on gx. Set the GUIX on gx configuration as shown below.

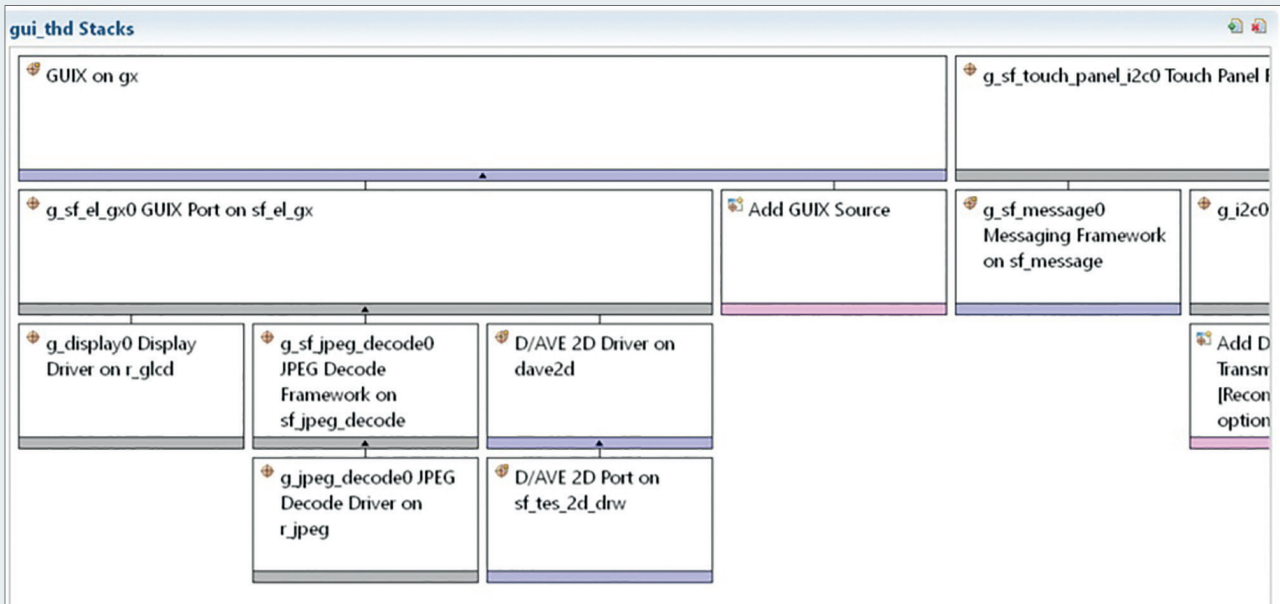


Fig. 170

Property	Value
Common	
Enable Synergy 2D Drawing Engine Support	Yes
Enable Synergy JPEG Support	Yes

Fig. 171

- Set the GUIX Port as follows

Property	Value
Common	
Parameter Checking	Default (BSP)
Module g_sf_el_gx0 GUIX Port on sf_el_gx	
Name	g_sf_el_gx0
Display Driver Configuration Inheritance	Inherit Graphics Screen 1
Name of User Callback function	NULL
Screen Rotation Angle(Clockwise)	0
GUIX Canvas Buffer (required if rotation angle is:	Not used
Size of JPEG Work Buffer (valid if JPEG hardware:	81920
Memory section for GUIX Canvas Buffer	sdram
Memory section for JPEG Work Buffer	bss

Fig. 172

- Configure the Touch Panel Framework as illustrated below.

Property	Value
▼ Common	
Parameter Checking	Default (BSP)
▼ Module g_display0 Display Driver on r_glcd	
Name	g_display0
Name of display callback function to be defined	NULL
Input - Panel clock source select	Internal clock(GLCDCLK)
Input - Graphics screen1	Used
Input - Graphics screen1 frame buffer name	fb_background
Input - Number of Graphics screen1 frame buff	2
Input - Section where Graphics screen1 frame b	sdram
Input - Graphics screen1 input horizontal size	240
Input - Graphics screen1 input vertical size	320
Input - Graphics screen1 input horizontal stride	256
Input - Graphics screen1 input format	16bits RGB565
Input - Graphics screen1 input line descending	Not used
Input - Graphics screen1 input lines repeat	Off
Input - Graphics screen1 input lines repeat time	0
Input - Graphics screen1 layer coordinate X	0
Input - Graphics screen1 layer coordinate Y	0
Input - Graphics screen1 layer background colc	255
Input - Graphics screen1 layer background colc	255
Input - Graphics screen1 layer background colc	255
Input - Graphics screen1 layer background colc	255
Input - Graphics screen1 layer fading control	None
Input - Graphics screen1 layer fade speed	0
Input - Graphics screen2	Not used
Input - Graphics screen2 frame buffer name	fb_foreground
Input - Number of Graphics screen2 frame buff	2
Input - Section where Graphics screen2 frame b	sdram
Input - Graphics screen2 input horizontal size	800
Input - Graphics screen2 input vertical size	480
Input - Graphics screen2 input horizontal stride	800
Input - Graphics screen2 input format	16bits RGB565
Input - Graphics screen2 line descending	Off

Fig. 173

Properties Problems

g_display0 Display Driver on r_glcd

Settings

Information

Input - Graphics screen2 input lines repeat	Off
Input - Graphics screen2 input lines repeat time	0
Input - Graphics screen2 layer coordinate X	0
Input - Graphics screen2 layer coordinate Y	0
Input - Graphics screen2 layer background colc	255
Input - Graphics screen2 layer background colc	255
Input - Graphics screen2 layer background colc	255
Input - Graphics screen2 layer background colc	255
Input - Graphics screen2 layer fading control	None
Input - Graphics screen2 layer fade speed	0
Output - Horizontal total cycles	320
Output - Horizontal active video cycles	240
Output - Horizontal back porch cycles	6
Output - Horizontal sync signal cycles	4
Output - Horizontal sync signal polarity	Low active
Output - Vertical total lines	328
Output - Vertical active video lines	320
Output - Vertical back porch lines	4
Output - Vertical sync signal lines	4
Output - Vertical sync signal polarity	Low active
Output - Format	16bits RGB565
Output - Endian	Little endian
Output - Color order	RGB
Output - Data Enable Signal Polarity	High active
Output - Sync edge	Rising edge
Output - Background color alpha channel	255
Output - Background color R channel	0
Output - Background color G channel	0
Output - Background color B channel	0
CLUT	Not used
CLUT - CLUT buffer size	256
TCON - Hsync pin select	LCD_TCON2
TCON - Vsync pin select	LCD_TCON1
TCON - DataEnable pin select	LCD_TCON0
TCON - Panel clock division ratio	1/24
Color correction - Brightness	Off

Fig. 174

Properties Problems

g_display0 Display Driver on r_glcd

Settings	Value
Color correction - Brightness R channel	512
Color correction - Brightness G channel	512
Color correction - Brightness B channel	512
Color correction - Contrast	Off
Color correction - Contrast(gain) R channel	128
Color correction - Contrast(gain) G channel	128
Color correction - Contrast(gain) B channel	128
Color correction - Gamma correction(Red)	Off
Color correction - Gamma gain R[0]	0
Color correction - Gamma gain R[1]	0
Color correction - Gamma gain R[2]	0
Color correction - Gamma gain R[3]	0
Color correction - Gamma gain R[4]	0
Color correction - Gamma gain R[5]	0
Color correction - Gamma gain R[6]	0
Color correction - Gamma gain R[7]	0
Color correction - Gamma gain R[8]	0
Color correction - Gamma gain R[9]	0
Color correction - Gamma gain R[10]	0
Color correction - Gamma gain R[11]	0
Color correction - Gamma gain R[12]	0
Color correction - Gamma gain R[13]	0
Color correction - Gamma gain R[14]	0
Color correction - Gamma gain R[15]	0
Color correction - Gamma threshold R[0]	0
Color correction - Gamma threshold R[1]	0
Color correction - Gamma threshold R[2]	0
Color correction - Gamma threshold R[3]	0
Color correction - Gamma threshold R[4]	0
Color correction - Gamma threshold R[5]	0
Color correction - Gamma threshold R[6]	0
Color correction - Gamma threshold R[7]	0
Color correction - Gamma threshold R[8]	0
Color correction - Gamma threshold R[9]	0
Color correction - Gamma threshold R[10]	0
Color correction - Gamma threshold R[11]	0

Fig. 175

Properties Problems

g_display0 Display Driver on r_glcd

Settings	Value
Color correction - Gamma threshold R[12]	0
Color correction - Gamma threshold R[13]	0
Color correction - Gamma threshold R[14]	0
Color correction - Gamma threshold R[15]	0
Color correction - Gamma correction(Green)	Off
Color correction - Gamma gain G[0]	0
Color correction - Gamma gain G[1]	0
Color correction - Gamma gain G[2]	0
Color correction - Gamma gain G[3]	0
Color correction - Gamma gain G[4]	0
Color correction - Gamma gain G[5]	0
Color correction - Gamma gain G[6]	0
Color correction - Gamma gain G[7]	0
Color correction - Gamma gain G[8]	0
Color correction - Gamma gain G[9]	0
Color correction - Gamma gain G[10]	0
Color correction - Gamma gain G[11]	0
Color correction - Gamma gain G[12]	0
Color correction - Gamma gain G[13]	0
Color correction - Gamma gain G[14]	0
Color correction - Gamma gain G[15]	0
Color correction - Gamma threshold G[0]	0
Color correction - Gamma threshold G[1]	0
Color correction - Gamma threshold G[2]	0
Color correction - Gamma threshold G[3]	0
Color correction - Gamma threshold G[4]	0
Color correction - Gamma threshold G[5]	0
Color correction - Gamma threshold G[6]	0
Color correction - Gamma threshold G[7]	0
Color correction - Gamma threshold G[8]	0
Color correction - Gamma threshold G[9]	0
Color correction - Gamma threshold G[10]	0
Color correction - Gamma threshold G[11]	0
Color correction - Gamma threshold G[12]	0
Color correction - Gamma threshold G[13]	0

Fig. 176

Settings	Information
Color correction - Gamma threshold G[14]	0
Color correction - Gamma threshold G[15]	0
Color correction - Gamma correction(Blue)	Off
Color correction - Gamma gain B[0]	0
Color correction - Gamma gain B[1]	0
Color correction - Gamma gain B[2]	0
Color correction - Gamma gain B[3]	0
Color correction - Gamma gain B[4]	0
Color correction - Gamma gain B[5]	0
Color correction - Gamma gain B[6]	0
Color correction - Gamma gain B[7]	0
Color correction - Gamma gain B[8]	0
Color correction - Gamma gain B[9]	0
Color correction - Gamma gain B[10]	0
Color correction - Gamma gain B[11]	0
Color correction - Gamma gain B[12]	0
Color correction - Gamma gain B[13]	0
Color correction - Gamma gain B[14]	0
Color correction - Gamma gain B[15]	0
Color correction - Gamma threshold B[0]	0
Color correction - Gamma threshold B[1]	0
Color correction - Gamma threshold B[2]	0
Color correction - Gamma threshold B[3]	0
Color correction - Gamma threshold B[4]	0
Color correction - Gamma threshold B[5]	0
Color correction - Gamma threshold B[6]	0
Color correction - Gamma threshold B[7]	0
Color correction - Gamma threshold B[8]	0
Color correction - Gamma threshold B[9]	0
Color correction - Gamma threshold B[10]	0
Color correction - Gamma threshold B[11]	0
Color correction - Gamma threshold B[12]	0
Color correction - Gamma threshold B[13]	0
Color correction - Gamma threshold B[14]	0
Color correction - Gamma threshold B[15]	0

Fig. 177

Dithering	Off
Dithering - Mode	Truncate
Dithering - Pattern A	Pattern 11
Dithering - Pattern B	Pattern 11
Dithering - Pattern C	Pattern 11
Dithering - Pattern D	Pattern 11
Misc - Correction Process Order	Brightness and Contrast then Gamma
Line Detect Interrupt Priority	Priority 3 (CM4: valid, CM0+: lowest - not valid if using ThreadX)
Underflow 1 Interrupt Priority	Priority 3 (CM4: valid, CM0+: lowest - not valid if using ThreadX)
Underflow 2 Interrupt Priority	Priority 3 (CM4: valid, CM0+: lowest - not valid if using ThreadX)

Fig. 178

Properties Problems

g_jpeg_decode0 JPEG Decode Driver on r_jpeg

Property	Value
▼ Common	
Parameter Checking	Default (BSP)
▼ Module g_jpeg_decode0 JPEG Decode Driver on r	
Name	g_jpeg_decode0
Byte Order for Input Data Format	Normal byte order (1)(2)(3)(4)(5)(6)(7)(8)
Byte Order for Output Data Format	Normal byte order (1)(2)(3)(4)(5)(6)(7)(8)
Output Data Color Format	Pixel Data RGB565 format
Alpha value to be applied to decoded pixel dat	255
Name of user callback function	NULL
Decompression Interrupt Priority	Priority 3 (CM4: valid, CM0+: lowest - not valid if using ThreadX)
Data Transfer Interrupt Priority	Priority 3 (CM4: valid, CM0+: lowest - not valid if using ThreadX)

Fig. 179

Properties Problems

g_jpeg_decode0 JPEG Decode Driver on r_jpeg

Property	Value
▼ Common	
Parameter Checking	Default (BSP)
▼ Module g_jpeg_decode0 JPEG Decode Driver on r	
Name	g_jpeg_decode0
Byte Order for Input Data Format	Normal byte order (1)(2)(3)(4)(5)(6)(7)(8)
Byte Order for Output Data Format	Normal byte order (1)(2)(3)(4)(5)(6)(7)(8)

Fig. 180

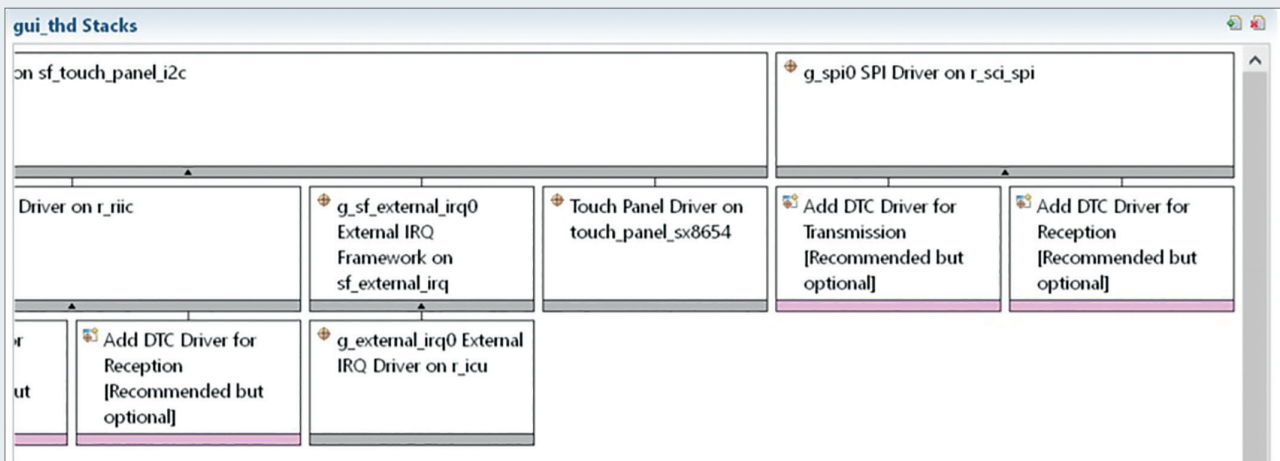
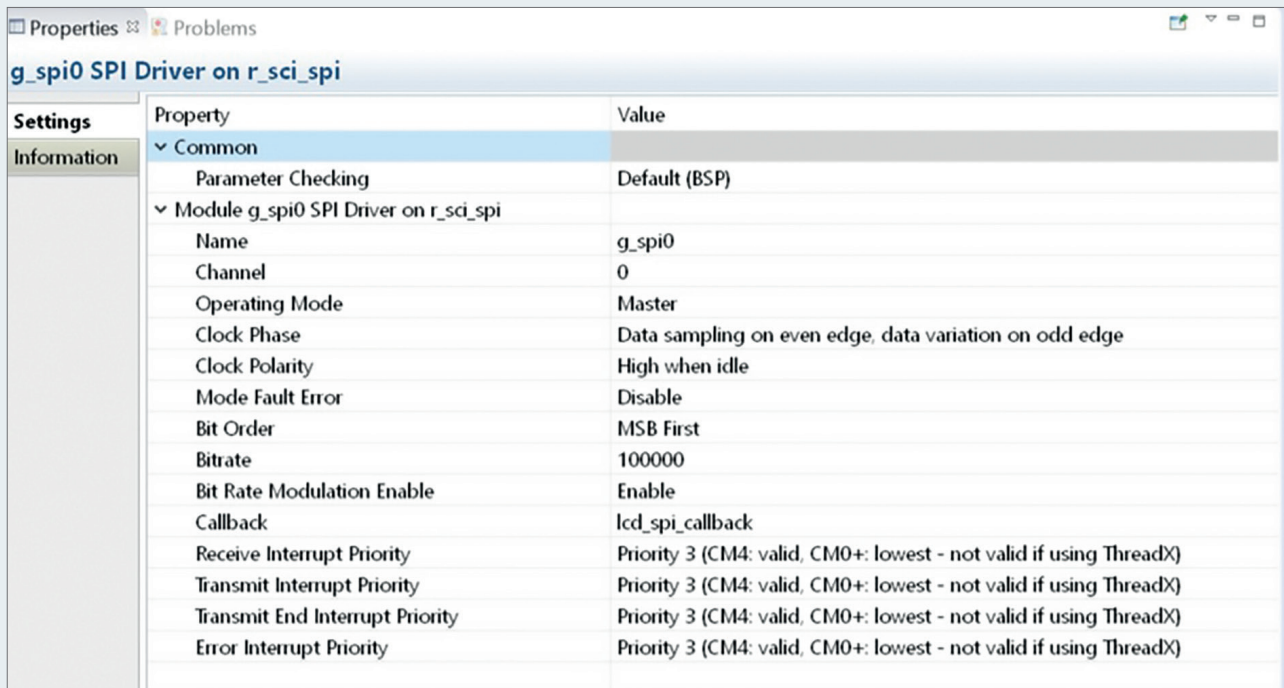


Fig. 181

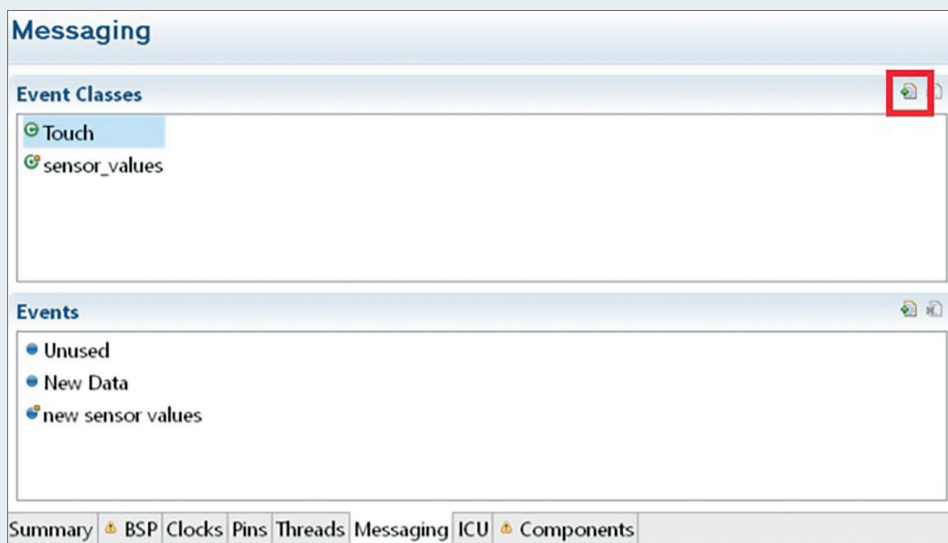
- Configure the stack as shown below. All these steps conclude the stack configuration for the GUI application.



Property	Value
Common	
Parameter Checking	Default (BSP)
Module g_spi0 SPI Driver on r_sci_spi	
Name	g_spi0
Channel	0
Operating Mode	Master
Clock Phase	Data sampling on even edge, data variation on odd edge
Clock Polarity	High when idle
Mode Fault Error	Disable
Bit Order	MSB First
Bitrate	100000
Bit Rate Modulation Enable	Enable
Callback	lcd_spi_callback
Receive Interrupt Priority	Priority 3 (CM4: valid, CM0+: lowest - not valid if using ThreadX)
Transmit Interrupt Priority	Priority 3 (CM4: valid, CM0+: lowest - not valid if using ThreadX)
Transmit End Interrupt Priority	Priority 3 (CM4: valid, CM0+: lowest - not valid if using ThreadX)
Error Interrupt Priority	Priority 3 (CM4: valid, CM0+: lowest - not valid if using ThreadX)

Fig. 182

- Now go to the messaging tab and add two event classes, Touch and sensor_values. Click on new event class and insert "Touch" and "sensor_values" into the pop-up window respectively. The other fields of the pop-up windows will be completed automatically.



Messaging

Event Classes

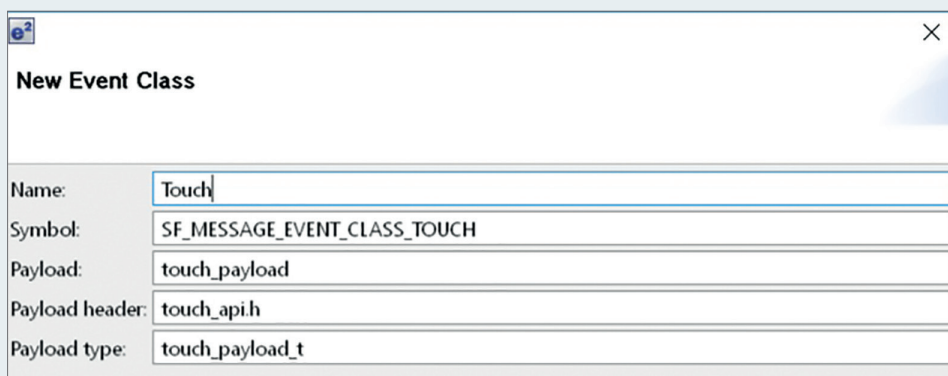
- Touch
- sensor_values

Events

- Unused
- New Data
- new sensor values

Summary | BSP | Clocks | Pins | Threads | Messaging | ICU | Components

Fig. 183



New Event Class

Name: Touch

Symbol: SF_MESSAGE_EVENT_CLASS_TOUCH

Payload: touch_payload

Payload header: touch_api.h

Payload type: touch_payload_t

Fig. 184

- For both event classes add the subscriber `gui_thd`.

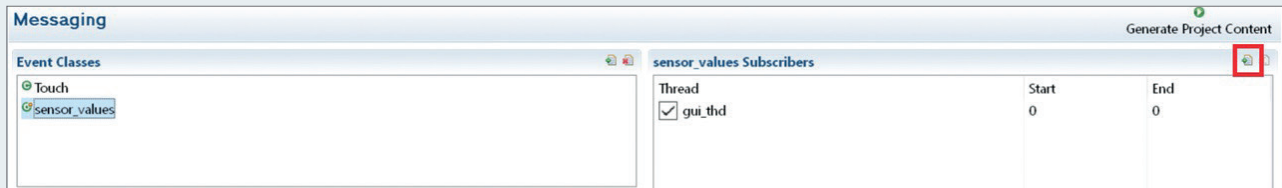


Fig. 185

- After the configuration insert the `gui_thd_files` folder into `src` folder in the project explorer.

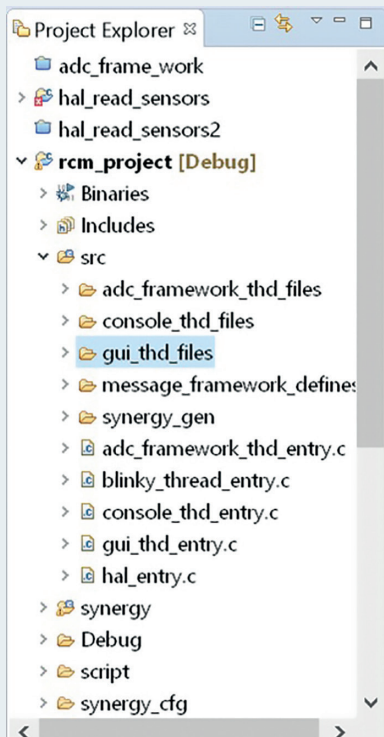


Fig. 186

- There should be no errors during building this project. Debugging the program should result into the LCD looking like designed in GUIX Studio™.
- Now we want to discuss the code used for the LCD to gain a major understanding. The files `lcd_setup.c` and `lcd.h` are necessary for initializing the lcd and providing the read/write LCD functions. When opening the `lcd_setup.c` one will notice the use of a semaphore that we declared in the spi driver stack.
- The `lcd_write` function is illustrated below. One can see that a semaphore is used after a write operation to synchronize with the end of the transmission. The spi callback is called when the write process is finished, so that the program can keep running again.

```

static void lcd_write(uint8_t cmd, char *data ,uint32_t len)
{
    ssp_err_t err;

    g_ioport_on_ioport.pinWrite(LCD_CMD, IOPORT_LEVEL_LOW);
    g_ioport_on_ioport.pinWrite(LCD_CS, IOPORT_LEVEL_LOW);

    err = g_spi0.p_api->write(g_spi0.p_ctrl, &cmd, 1, SPI_BIT_WIDTH_8_BITS);
    if (SSP_SUCCESS != err)
    {
        while(1);
    }

    /* wait for spi callback for write end signal */
    tx_semaphore_get(&lcd_semaphore, TX_WAIT_FOREVER);

    if (len)
    {
        g_ioport_on_ioport.pinWrite(LCD_CMD, IOPORT_LEVEL_HIGH);

        err = g_spi0.p_api->write(g_spi0.p_ctrl, (void const *)data, len, SPI_BIT_WIDTH_8_BITS);
        if (SSP_SUCCESS != err)
        {
            while(1);
        }

        /* wait for spi callback for write end signal */
        tx_semaphore_get(&lcd_semaphore, TX_WAIT_FOREVER);
    }
    g_ioport_on_ioport.pinWrite(LCD_CS, IOPORT_LEVEL_HIGH);
}

```

- The `guiapp_event_handlers.c` defines what should happen when the buttons on the windows are pressed. In this case change from one window to another. The code below illustrates what the entry window does when the icon button is clicked. Note that the entry window handler is declared in GUIX Studio™ in the ENTRY windows properties. This is done for the other windows as well.

```

UINT entry_window_handler(GX_WINDOW *widget, GX_EVENT *event_ptr)
{
    switch (event_ptr->gx_event_type)
    {
        case GX_SIGNAL(ID_B_ENTRY, GX_EVENT_CLICKED):
            show_window((GX_WINDOW*)&APP_SELECT, (GX_WIDGET*)widget, true);
            break;
        default:
            gx_window_event_process(widget, event_ptr);
            break;
    }
    return result;
}

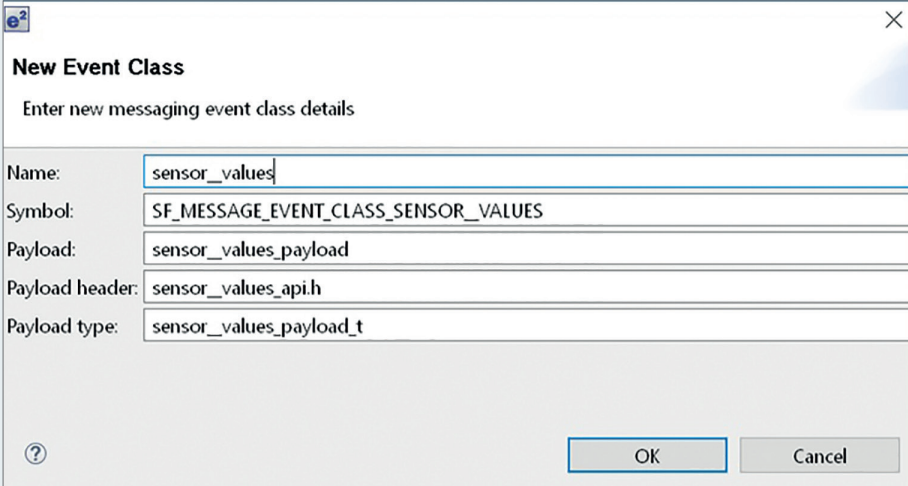
```

- The `gui_def.c` is the most complex part that initializes the LCD and uses within the working loop the message framework to receive Touch Panel events. We will discuss the message framework in the final step of the project in order to transmit the sensor data to the LCD via message framework, which will finalize our project

12.10. THE MESSAGE FRAMEWORK

In this final part the project utilizes the message framework to send sensor data to the GUI thread. The message framework is a powerful tool that enables to build complex projects. Though it is a very useful tool, one needs a basic understanding on how it works to utilize it correctly.

- Import the GUIX™ project and rename it to `rcm_project`.
- Open up the synergy configuration by double clicking on the `configuration.xml` file. Open up the `gui_thd` stack window and go to the Touch Panel Framework stack. The `g_sf_message0` Messaging Framework stack has been auto-included with the Touch Panel Framework. In the Messaging Tab the Event and Event Classes for the LCD were auto-included as well. In this case we want to make use of the existing message framework and add a new event class which carries our sensor data.



New Event Class
Enter new messaging event class details

Name:

Symbol:

Payload:

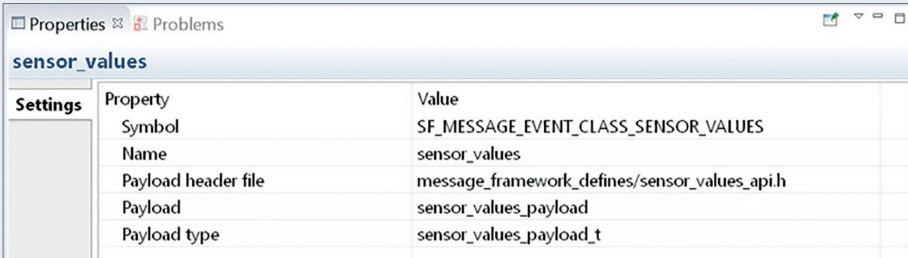
Payload header:

Payload type:

OK Cancel

Fig. 187

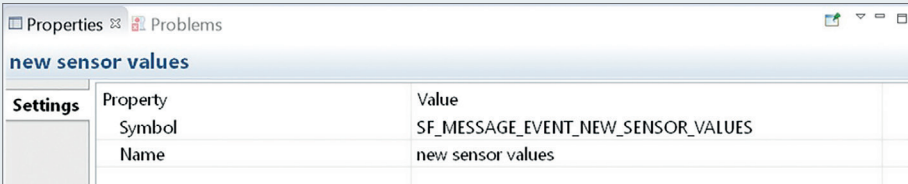
- The definitions for the events are stored in the `sensor_values_api.h`. Messaging Framework configurator exports event/class names into `sf_message_port.h` (in `synergy_cfg/ssp_cfg/framework`). Payload structures are included directly from the header provided by the user. In this case the path to the header file needs to be changed in the property window of the new `sensor_values` event class.



Property	Value
Symbol	SF_MESSAGE_EVENT_CLASS_SENSOR_VALUES
Name	sensor_values
Payload header file	message_framework_defines/sensor_values_api.h
Payload	sensor_values_payload
Payload type	sensor_values_payload_t

Fig. 188

- Next we add a new event under "Events" for the newly created class called "new sensor values" and assign `gui_thd` to the subscribers to the sensor values event class.



Property	Value
Symbol	SF_MESSAGE_EVENT_NEW_SENSOR_VALUES
Name	new sensor values

Fig. 189

- This step finalizes the configuration and the code can be generated.
- Next is setting up the producer of our subscriber list, which will be in this case the `adc_framework_thd_entry.c`. We add the `dth11` sensor values reading to this file, that we developed earlier to have all sensor data read in a single file and send to the message framework. The code is posted below and will be explained step by step.


```

#include "adc_framework_thd.h"
#include "adc_framework_thd_files/dth11_protocol_sm.h"

/* global variables */
uint16_t adc_water_level;
uint16_t adc_brightness;
uint8_t dth11_humidity;
uint8_t dth11_temperature;

/* callback function */
void g_adc_framework_user_callback(sf_adc_periodic_callback_args_t * p_args){

    uint32_t index = p_args->buffer_index;

    adc_brightness = (uint16_t)(g_user_buffer[index] + g_user_buffer[index+2] + g_user_buffer[index+4]) / 3;
    adc_water_level = (uint16_t)(g_user_buffer[index+1] + g_user_buffer[index+3] + g_user_buffer[index+5]) / 3;
}

/* adc_framework_thd entry function */
void adc_framework_thd_entry(void)
{
    /* variables */
    ssp_err_t err;
    uint8_t cnt = 0;

    /* address of allocated buffer */
    sf_message_header_t* p_buffer;
    /* mark keep option - does not release buffer */
    sf_message_acquire_cfg_t acquire_cfg = { .buffer_keep = true };
    /* set priority normal no call backs when acknowledged */
    sf_message_post_cfg_t post_cfg =
    {
        .priority = SF_MESSAGE_PRIORITY_NORMAL,
        .p_callback = NULL,
    };
    /* error message post */
    sf_message_post_err_t err_post;
    /* our pay load variable */
    sensor_values_payload_t* p_payload;

    /* get buffer */
    err = g_sf_message0.p_api->bufferAcquire(g_sf_message0.p_ctrl, &p_buffer, &acquire_cfg, 300);
    if( err != SSP_SUCCESS) for(;;);

    p_payload = (sensor_values_payload_t*) p_buffer;
    p_payload->header.event_b.class_code = SF_MESSAGE_EVENT_CLASS_SENSOR_VALUES;
    p_payload->header.event_b.class_instance = 0;
    p_payload->header.event_b.code = SF_MESSAGE_EVENT_NEW_SENSOR_VALUES;

    /* start the scan, since auto start is off */
    err = g_sf_adc_periodic0.p_api->start(g_sf_adc_periodic0.p_ctrl);
    if( err != SSP_SUCCESS) for(;;);

    err = dth11_protocol_init();
    if( err != SSP_SUCCESS) for(;;);

    while (1)
    {
        /* update sensor values */
        if(cnt == 0)
        {
            err = dth11_protocol(&dth11_humidity, &dth11_temperature);
            if( SSP_SUCCESS == err)
            {
                /* error handling
            }
            /* wait 1 second to get dth11 data */
            cnt = 6;
        }
        --cnt;

        p_payload->temperature = (uint16_t) dth11_temperature;
        p_payload->humidity = (uint16_t) dth11_humidity;
        p_payload->water_level = adc_water_level;
        p_payload->brightness = adc_brightness;

        /* update lcd */
        err = g_sf_message0.p_api->post(g_sf_message0.p_ctrl,
            (sf_message_header_t*)p_payload,
            &post_cfg,
            &err_post,
            300);
        if( err != SSP_SUCCESS) for(;;);

        /* wait for 200 ms */
        tx_thread_sleep(20);
    }
}

```

- The first step in using the message framework is to allocate a buffer that has sufficient RAM to store our data. Create a pointer of type `sf_message_header_t` that will point to the allocated buffer.
- The next step is to define the allocation configuration. In this case set the buffer to stay after a release call, so that we can reuse the same buffer area and save overhead. The priority setting is set to normal and there is no need to call any callback function, that is why it is set to NULL.
- Define a pointer to the payload that will be used later on.
- The `bufferAcquire` function will wait for a duration of 300 ticks to acquire the data.

```

/* address of allocated buffer */
sf_message_header_t* p_buffer;
/* mark keep option - does not release buffer */
sf_message_acquire_cfg_t acquire_cfg = { .buffer_keep = true };
/* set priority normal no call backs when acknowledged */
sf_message_post_cfg_t post_cfg =
{
    .priority = SF_MESSAGE_PRIORITY_NORMAL,
    .p_callback = NULL,
};
/* error message post */
sf_message_post_err_t err_post;
/* our payload variable */
sensor_values_payload_t* p_payload;

/* get buffer */
err = g_sf_message0.p_api->bufferAcquire(g_sf_message0.p_ctrl, &p_buffer, &acquire_cfg, 300);
if( err != SSP_SUCCESS) for(;;);

```

- The function will write an address to `p_buffer` where we can save our data. Therefore we cast the `p_buffer` to `sensor_values_payload_t` and write it to `p_payload` and fill in the `event_b` data, that distinguishes our message from others.

```

p_payload = (sensor_values_payload_t*) p_buffer;
p_payload->header.event_b.class_code = SF_MESSAGE_EVENT_CLASS_SENSOR_VALUES;
p_payload->header.event_b.class_instance = 0;
p_payload->header.event_b.code = SF_MESSAGE_EVENT_NEW_SENSOR_VALUES;

```

- Within the endless loop of the thread we assign the data and post the message to the message framework, which will handle our message. The message gets posted with a rate of 5 Hz.

```

p_payload->temperature = (uint16_t) dth11_temperature;
p_payload->humidity = (uint16_t) dth11_humidity;
p_payload->water_level = adc_water_level;
p_payload->brightness = adc_brightness;

/* update lcd */
err = g_sf_message0.p_api->post(g_sf_message0.p_ctrl,
    (sf_message_header_t*)p_payload,
    &post_cfg,
    &err_post,
    300);
if( err != SSP_SUCCESS) for(;;);

```

- This finalizes setting up the message producer. Set up the `gui_thd` to receive the message and process it accordingly.
- Open the `gui_def.c` and compare the adjustments that were made to the project before and the code posted below.

```

/* here starts the working loop */
while(1)
{
    bool new_gui_event = false;

    /* wait for the message framework to post a touch event */
    err = g_sf_message0.p_api->pend(g_sf_message0.p_ctrl, &gui_thd_message_queue, (sf_message_header_t **) &p_message,
                                   TX_WAIT_FOREVER);

    if (err)
    {
        /* TODO: Handle error. */
    }

    switch (p_message->event_b.class_code)
    {
    case SF_MESSAGE_EVENT_CLASS_TOUCH:
    {
        switch (p_message->event_b.code)
        {
            case SF_MESSAGE_EVENT_NEW_DATA:
            {
                /* Translate an SSP touch event into a GUIX event */
                new_gui_event = ssp_touch_to_guix((sf_touch_panel_payload_t*)p_message, &g_gx_event);
            }
            default:
                break;
        }

        /* Message is processed, so release buffer. */
        err = g_sf_message0.p_api->bufferRelease(g_sf_message0.p_ctrl, (sf_message_header_t *) p_message,
                                                SF_MESSAGE_RELEASE_OPTION_FORCED_RELEASE);
        /* standard procedure pend and the release buffer */

        break;
    }
    /* add data receive message */
    case SF_MESSAGE_EVENT_CLASS_SENSOR_VALUES:
    {
        switch(p_message->event_b.code)
        {
            /* output last values in queue to the console */
            case SF_MESSAGE_EVENT_NEW_SENSOR_VALUES:
            {
                new_gui_event = sensor_values_message_sm((sensor_values_payload_t*)p_message, &g_gx_event);
                break;
            }
            default: /* do nothing */
                break;
        }

        /* Message is processed, so release buffer. */
        err = g_sf_message0.p_api->bufferRelease(g_sf_message0.p_ctrl, (sf_message_header_t *) p_message,
                                                SF_MESSAGE_RELEASE_OPTION_NONE);
        /* standard procedure pend and the release buffer */

        break;
    }
    default:
        break;
    }

    if (err)
    {
        /* TODO: Handle error. */
    }

    /* Post message. */
    if (new_gui_event) {
        gx_system_event_send(&g_gx_event);
    }
}

```

- The only part that changed is the working loop. It now contains the sensor values event class and the respective `bufferRelease()` function. If a sensor value event is send, then the `sensor_values_message_sm` is called. This function processes the event and updates the LCD to show the sensor values. The code of the function is illustrated below.

```
static bool sensor_values_message_sm(sensor_values_payload_t* p_payload, GX_EVENT * gx_event)
{
    bool send_event = true;
    /* update text */

    /* build temperature string */
    memset((void*) str_temperature, 0, sizeof(str_temperature));
    uint_2_str((uint16_t)p_payload->temperature, (char*) str_conversion_temp);
    strcat((char*)str_temperature, (char*)str_conversion_temp);
    strcat((char*)str_temperature, " *C");
    /* update */
    update_rcm_prompt_id((USHORT) ID_P_TEMP, (char*) str_temperature);

    /* build humidity string */
    memset((void*) str_humidity, 0, sizeof(str_humidity));
    uint_2_str((uint16_t)p_payload->humidity, (char*) str_conversion_temp);
    strcat((char*)str_humidity, (char*) str_conversion_temp);
    strcat((char*)str_humidity, " %RH");
    /* update */
    update_rcm_prompt_id((USHORT) ID_P_HUMI, (char*) str_humidity);

    /* build brightness string */
    memset((void*) str_brightness, 0, sizeof(str_brightness));
    /* note that the brightness sensor is low for bright places, so we need to converse it */
    uint_2_str((uint16_t)((200 - p_payload->brightness)/2), (char*) str_conversion_temp);
    strcat((char*)str_brightness, (char*) str_conversion_temp);
    strcat((char*)str_brightness, " %RB");
    /* update */
    update_rcm_prompt_id((USHORT) ID_P_BRIGHT, (char*) str_brightness);

    /* build humidity string */
    memset((void*) str_water_level, 0, sizeof(str_water_level));
    strcat((char*) str_water_level, " ");
    uint_2_str((uint16_t)(p_payload->water_level * 1.7 ), (char*) str_conversion_temp);
    strcat((char*) str_water_level, (char*)str_conversion_temp);
    strcat((char*) str_water_level, " %RWL");
    /* update */
    update_rcm_prompt_id((USHORT) ID_P_WATER_LEV, (char*) str_water_level);

    /* send event to draw window */
    gx_event->gx_event_type = GX_EVENT_REDRAW;
    return send_event;
}

static void update_rcm_prompt_id( USHORT id, char* text)
{
    GX_PROMPT * p_prompt = NULL;
    /* note ROOM_MONITOR was the name of our window widget */
    ssp_err_t err = gx_widget_find(&ROOM_MONITOR, id, GX_SEARCH_DEPTH_INFINITE, (GX_WIDGET**) &p_prompt);
    if (TX_SUCCESS == err)
    {
        gx_prompt_text_set(p_prompt, text);
    }
}
}
```

- The function creates the needed strings similar to the console application and calls the `update_rcm_project_prompt_id()` function. This function sets the strings to the prompts, that is done to all 4 sensor values respectively. At the end, we need to send a redraw event, in order to update the LCD screen.

The room condition monitor application is finished.

REFERENCES

1. S7G2 User's Manual: Microcontrollers, Rev.1.20, Aug 2016, Renesas Electronics
r01um0001eu0120-synergy-s7g2.pdf
2. Renesas Synergy Software Package (SSP v1.2.0) User's Manual, v1.00, Feb 2017
3. Basics of the Renesas Synergy Platform, Richard Oed, 2016.11
4. Renesas Synergy Software Package (SSP v1.2.0), Datasheet, SSP Version 1.34, Mar 2017
r01ds0272eu0134-synergy-ssp-120-datasheet.pdf
5. Cortex-M4 Devices Generic User Guide, 2010, ARM DUI0553A_cortex_m4_dgug.pdf
6. Renesas Synergy Starter Kit SK-S7G2 User's Manual, Rev. 1.00, Oct. 2015
r12um0004eu0100_synergy_sk_s7g2.pdf
7. e²studio v5.2.1.016, Rev.1.10 Dec 2016
r11an0052eu0110-e²studioisde-v521016-relnote.pdf
8. Renesas Synergy Software Package (SSP) User's Manual, v1.2.0-b.1, Rev.00.96 November 2016
r01us0171eu0096_synergy_esp.pdf
9. Renesas Synergy Software Package (SSP) Datasheet, SSP v1.2.0, Rev. 1.34, Mar 2017
r01ds0272eu0134-synergy-ssp-120-datasheet.pdf
10. Inside the Synergy Software Platform, Lake Garda, April 2016
2_3 Inside the Synergy Software Platform plus SSP in Action.pdf
11. Real-Time Embedded Multithreading Using ThreadX: Third Edition, Edward L. Lamie, June 2015
12. Creating Fast, Responsive and Energy-Efficient Embedded Systems using the Renesas RL78 Microcontroller, Alexander G. Dean and James M. Conrad, March 2012
13. GUIX Renesas Synergy Platform User Guide, Rev.5.30, Aug 2016
r11um0003eu0530-synergy-guix.pdf
14. GUIX Studio Renesas Synergy Platform User Guide, Rev.5.30, May 2016
r11um0002eu0530_synergy_guix_studio.pdf
15. FileX Renesas Synergy Platform User's Manual: Software, Rev.5.0
r11um0001eu0500_synergy_filex.pdf
16. Seamless integration of communication protocols, Lake Garda, April 2016
12 Seamless integration of communication protocols Lecture.pdf
17. NetX Renesas Synergy Platform User Guide, Rev.5.90, Aug 2016
18. NetX Duo Renesas Synergy Platform User Guide, Rev.5.90, Aug 2016
19. USBX Host Stack Renesas Synergy Platform User's Manual: Software, Rev.5
20. USBX Device Stack Renesas Synergy Platform User's Manual: Software, Rev.5
21. <https://synergycastle.renesas.com/>

INDEX

A

Actuators 08
ADAS 06, 68
ADC 14, 20
ADC framework 60
API 10, 27, 36, 39, 40, 41, 43, 44, 45, 47, 54, 58, 62, 64, 65, 74, 77, 96, 99, 115
Application Examples 80
Application Framework 58
ARM 15
automation pyramid 06

B

BSP 35, 90

C

CAN 13, 20, 71
CANopen 71, 82
CMSIS 61
component-based software architecture 40
CPS 07
CPU 10, 11, 12, 13, 15, 18, 19, 23, 25, 32, 47, 48, 49, 50, 51, 52, 53, 86, 100
CRC 74
CSMA/CD 74
CTSU 24
Cyber Physical System 07

D

DAC 14, 23
deadlock 54
deadly embrace 54
DSC 10
DSP 10, 61

E

e² studio 30
Eclipse 30
ECU 48, 68
Embedded Systems 07

encryption 23, 62
Ethernet 21, 73
Ethernet MAC 21
Ethernet PHY 21

F

FileX® 63
FPGA 10
FPU 16
FTP 74
Functional Libraries 58

G

GCC compiler 39
GPIO 13
GUI 66, 122
GUIX Studio™ 66, 122
GUIX® 65
GUIX™ 122

H

HAL 40, 42, 43, 104
Hard real-time 49
HLD 44
HMD 24
HTTP 70, 74

I

I/O Ports 12
I²C 72
IDE 29
IEEE 802.3 21
Industry 4.0 06, 07, 68, 85
Internet of Things 06, 74, 85
Interrupt 14
IoT 06, 08, 10, 18, 19, 20, 21, 23, 24, 28, 47, 54, 55, 58, 60, 62, 63, 66, 67, 68, 72, 73, 85
IPv4 74
IPv6 74
ISDE 29
ISR 14, 51

L

LAN 21
LLD 44

M

MAC address 73
MEMS 9
Microcontroller 11
Middleware 63
MII 21
Modular software 41
MPU 20
mutex 53

N

NetX™ 74
NetX Duo™ 74
NVIC 20

O

operating system 47
OSI model 69

P

PCB 26
perspective 31
PMD 26
POP3 74
preemption-threshold 53
preemptive scheduling 51
preemption inheritance 53
priority inversion 53
product examples 80
PWM 14

Q

QSA 79

R

real-time 47, 48

Renesas Synergy Gallery 81
RISC 15
RTOS 47, 49, 96

S

S1 series 18
S3 series 18
S5 series 19
S7 series 19
S7G2 MCU 20
semaphore 53
Sensors 08
smart manual 31
SMTP 70, 74
soft real-time 48
SSP 28, 29, 30, 31, 35, 41, 43, 44, 50, 54, 55, 57, 59, 60, 61, 62, 63, 65, 66, 67, 71, 72, 74, 76, 77, 83, 87, 96, 98, 101
Starter Kit 26, 87
synchronization of threads 52
Synergy 05, 08, 18, 19, 27, 28, 30, 31, 35, 36, 39, 40, 43, 44, 47, 57, 63, 65, 66, 71, 76, 77, 78, 83, 85, 86, 87, 90, 95, 96, 110, 112

T

TCP/IP 70, 74
Telnet 74
thread 50
thread starvation 53
ThreadX® 54, 64
Timer 14, 22
TraceX® 57

U

UART 20
UDP 74
USB 21, 76
USBX™ 76

V

VSA 79

W

WAN 21

X

X-Ware suite 63

Before purchasing or using any Renesas Electronics products listed herein, please refer to the latest product manual and/or data sheet in advance.

Renesas Electronics Europe

www.renesas.com

