Linux-based Digital Frame System

Example Code Introduction for 32-bit NuMicro® Family

Document Information

Application	This example implements an application design on a Linux platform based on a digital photo frame system. The system includes a web server, allowing users to upload or update photos through a web browser phone. It offers a simple and user-friendly way to manage and update photos.
BSP Version	Linux-5.10.x
Hardware	NuMaker-HMI-N9H30

The information described in this document is the exclusive intellectual property of Nuvoton Technology Corporation and shall not be reproduced without permission from Nuvoton.

Nuvoton is providing this document only for reference purposes of NuMicro microcontroller and microprocessor based system design. Nuvoton assumes no responsibility for errors or omissions.

All data and specifications are subject to change without notice.

For additional information or questions, please contact: Nuvoton Technology Corporation.

www.nuvoton.com

1. Overview

This example is designed to meet the needs of overseas markets and demonstrates how to quickly implement a display application in a Linux environment. Using the NuMaker-HMI-N9H30 development platform, it takes an electronic photo frame as an example and gradually completes both GUI design and function integration. By leveraging the capabilities of the operating system, it also showcases the Web Server feature, enabling users to upload or update photos via a web browser. Photo uploads are fully supported on mobile phones and tablets as well.

This example is based on Nuvoton's GUI development platform, the NuMaker-HMI-N9H30 board, which integrates a 7-inch TFT-LCD resistive touch panel with a resolution of 800×480. Before proceeding, the Linux development environment for the NuMaker-HMI-N9H30 must be properly set up.

For configuration details, please refer to the *NUC9x0 Linux Environment on VMware User Manual* for the ARM9 series microprocessors.

2. Code Description

The overall framework of the digital frame system is illustrated below. It consists of two major parts:

Part 1 — Display Core

The main program continuously reads image files from the photo directory and renders them on the screen as a slideshow. This handles the core display functionality of the system.

Part 2 — Image Update via Web Server

A built-in Web Server enables users to upload new photos to the photo directory through a web browser or a mobile device over ethernet. Before uploading, images are automatically resized and rotated on the browser side to match the display resolution. This ensures the images can be displayed directly on the Linux system without any additional post-processing.

By combining these two mechanisms, the system provides a digital frame system solution that supports real-time image updates and seamless playback.

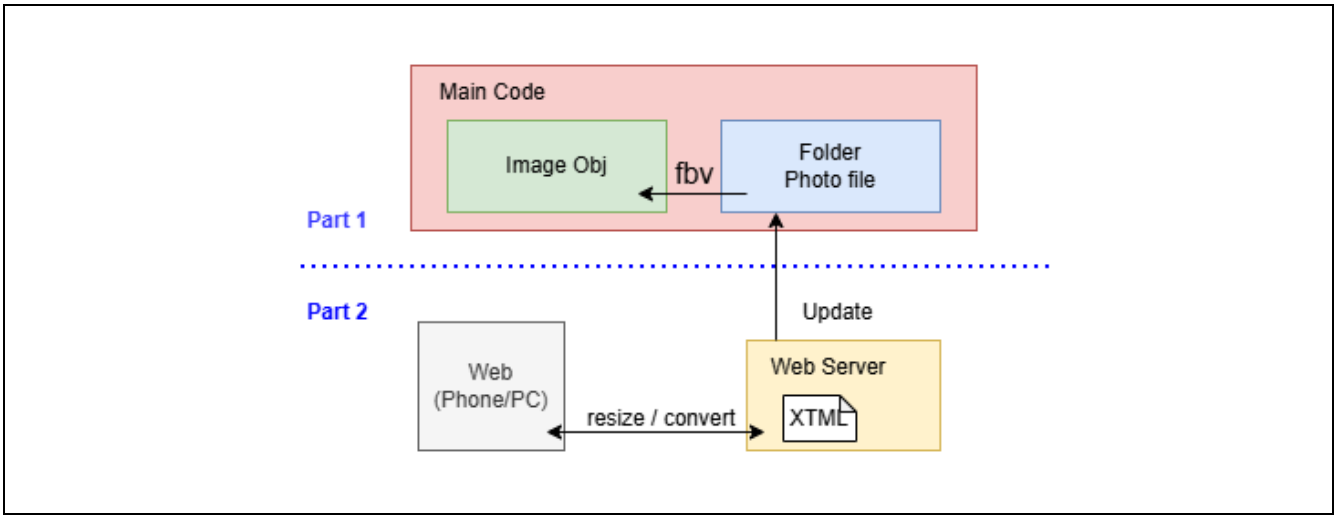

Figure 2-1 Electronic Photo Frame System Framework

For how to build the root file system in Linux, please refer to the *Linux 5.10 RootFS Build Guide*.

This chapter explains how to implement the photo update function within the Linux/Buildroot environment. In Buildroot menuconfig, enable the following required packages (you may use the “/” key to search for package names):

- **BR2_PACKAGE_DHCPCD**

  Used to obtain an IP address via DHCP. Handles automatic network configuration.

  Location :

  Target packages --- >

Networking applications --- >

[*] dhcpc

- **BR2_PACKAGE_NGINX**
  A lightweight web server that provides HTTP services for photo uploading and management.

  Location:

  Target packages --- >

  Networking applications --- >

  [*] nginx

- **BR2_PACKAGE_FBV**

  A simple framebuffer-based image viewer that can display JPG / PNG / BMP files directly on the Linux framebuffer.

  Location:

  Target packages --- >

  Graphic libraries and applications (graphic/text) --- >

  [*] fbv

- **BR2_PACKAGE_LIGHTTPD**

  A lightweight web server similar to Nginx, but with even lower resource usage — suitable for embedded systems. It can be used to provide HTTP services such as file uploads, static web pages, and REST APIs.

  Location:

  Target packages --- >

  Networking applications --- >

  [*] lighttpd

Part 1: Main Program

The main application continuously scans the */opt/photo* directory for image files with the *.png* extension. Each detected image is displayed on the screen in sequence using the **fbv** command, creating a slideshow effect. To ensure system stability, the program also releases the memory allocated by **scandir()** to prevent memory leaks. This approach provides a lightweight and efficient implementation that can serve as the core of an electronic photo frame or a basic display system.

Note that the program only checks image files with the *.png* extension. This is because all uploaded photos are restricted to be saved in PNG format, so the application only needs to process files with the *.png* extension.

```
int main(void)
{
    struct dirent **namelist;
    int count;
    char fullpath[CMD_MAX_LEN];
    char cmd[CMD_MAX_LEN];

    while (1) {
        // Scan the photo directory and filter only *.png files.
        // 'filter_png' will return only files ending with .png
        // 'cmp_png' will sort them by filename or timestamp depending on your
implementation.
        count = scandir(PHOTO_DIR, &namelist, filter_png, cmp_png);
        if (count < 0) {
            perror("scandir failed");   // Print error if directory scan fails
            return -1;
        }

        // If no PNG files are found, wait 1 second and try again.
        if (count == 0) {
            printf("No PNG files found in %s\n", PHOTO_DIR);
            sleep(1);
            continue;
        }

        // Display each PNG file sequentially in sorted order.
        for (int i = 0; i < count; i++) {

            // Build the full file path: PHOTO_DIR + "/" + filename
            snprintf(fullpath, sizeof(fullpath), "%s/%s",
                     PHOTO_DIR, namelist[i]->d_name);

            // Use fbv (framebuffer viewer) to display the image.
            // Options:
            //   -a : automatically zoom to best fit
            //   -i : auto exit after display
            // Redirect all output to /dev/null to keep console clean.
            snprintf(cmd, sizeof(cmd), "fbv -a -i %s > /dev/null 2>&1", fullpath);
            system(cmd);   // Execute the command

            // Free each allocated dirent entry returned by scandir()
            free(namelist[i]);


            usleep(mseconds * 100);
        }


        sleep(3);

        // Free the namelist array allocated by scandir()
        free(namelist);
```

```
    }

    return 0;
}
```

Part 2: Web Server Process

This section handles the workflow of the web server. The *index.html* page defines the user interface that appears in the browser. Through Wi-Fi/Ethernet, users can upload photos to the *photo* directory using either a desktop browser or a mobile device. Prior to uploading, image scaling and rotation are conducted on the browser side to ensure that the files are properly formatted and compatible with the display system.

The user upload page layout is shown as follows:

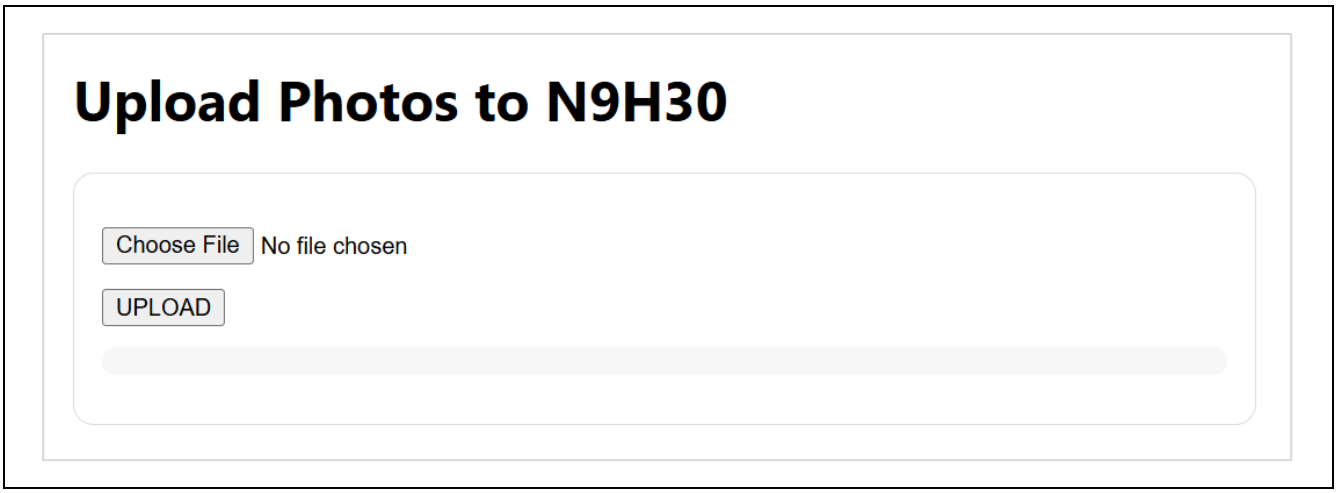

Figure 2-2 Web Display Page

The layout of the **HTML** webpage is planned as follows:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">

<!--
  Make layout responsive on mobile devices.
  Without this line, the UI will appear zoomed-in on smartphones.
-->
<meta name="viewport" content="width=device-width, initial-scale=1">

<title>Photo Upload</title>

<style>
  /*
   * ===== Global Page Layout =====
   * Use system fonts for better performance.
   * Max width is set so content remains readable on large displays.
   */
  body {
    font-family: system-ui, -apple-system, Segoe UI, Roboto, Arial;
```

```
    max-width: 680px;           /* Limit UI width for better readability */
    margin: 40px auto;          /* Center vertically on desktop */
    padding: 0 16px;
  }

  /* Simple card-style container with soft rounded borders */
  .card {
    border: 1px solid #ddd;
    border-radius: 12px;
    padding: 16px;
  }

  /* Maintain consistent vertical spacing between each UI section */
  .row { margin: 12px 0; }

  /* Upload progress bar (hidden before upload starts) */
  progress {
    width: 100%;
    height: 16px;
  }

  /*
   * Status display area (upload result / error message).
   * `white-space: pre-wrap` keeps line breaks in log output.
   */
  #log {
    white-space: pre-wrap;
    background: #f7f7f7;
    padding: 8px;
    border-radius: 8px;
  }
</style>
</head>

<body>
<h1>Upload Photos to N9H30</h1>

<div class="card">
  <!-- File picker: only accept image formats -->
  <div class="row">
    <input id="file" type="file" accept="image/*">
  </div>

  <!-- Upload trigger button -->
  <div class="row">
    <button id="btn">UPLOAD</button>
  </div>

  <!-- Progress bar: hidden until upload begins -->
  <div class="row">
    <progress id="bar" max="100" value="0" hidden></progress>
  </div>

  <!-- Upload result / status message -->
  <div class="row" id="log"></div>
</div>
```

loadBitmap() Function in the Web Page:

The **loadBitmap()** function is responsible for loading the image file selected via the **<input type="file">** element. It automatically applies the **EXIF** rotation metadata to ensure that photos taken with mobile devices are displayed correctly—preventing issues such as upside-down or sideways orientation.

```
/* ① Load image and apply EXIF orientation (if supported by browser) */
async function loadBitmap(file) {
  if ('createImageBitmap' in window) {
    try {
      return await createImageBitmap(file, { imageOrientation: 'from-image' });
    } catch (e) { /* Fallback if failed */ }
  }
  // Alternative method using Object URL
  const url = URL.createObjectURL(file);
  try {
    const img = new Image();
    img.decoding = 'async';
    img.src = url;
    await img.decode();
    return img;
  } finally {
    // URL will be revoked after drawing on canvas
  }
}
```

resizeToCanvas() Function:

The purpose of the **resizeToCanvas()** function is to scale the loaded image according to the screen resolution while maintaining its original aspect ratio. The image is drawn onto a canvas of fixed size, avoiding distortion. If any blank areas remain after scaling, they are filled with a specified background color, and the image is centered. This ensures that the final output matches the 800×480 display resolution exactly, reducing the image processing workload on the embedded Linux system.

```
/* ② Resize image to 800×480 (centered, keep aspect ratio, no distortion) */
function resizeToCanvas(bitmap, { maxW, maxH, background = '#000', upsize = true, mode =
'contain' }) {
  const iw = bitmap.width, ih = bitmap.height;
  let scale = mode === 'cover' ? Math.max(maxW/iw, maxH/ih) : Math.min(maxW/iw, maxH/ih);
  if (!upsize) scale = Math.min(scale, 1);
  const dw = Math.round(iw * scale);
  const dh = Math.round(ih * scale);

  // Create canvas buffer
  const canvas = document.createElement('canvas');
  canvas.width = maxW;
  canvas.height = maxH;

  const ctx = canvas.getContext('2d');
  ctx.fillStyle = background;
  ctx.fillRect(0, 0, maxW, maxH);
  ctx.imageSmoothingQuality = 'high';
```

```
  // Calculate offset (center image)
  const dx = Math.floor((maxW - dw) / 2);
  const dy = Math.floor((maxH - dh) / 2);
  ctx.drawImage(bitmap, dx, dy, dw, dh);

  return canvas;
}
```

canvasToBlob() Function:

The **canvasToBlob()** function converts the final rendered content into a Blob-format image file. It restricts the output to PNG format and allows the compression quality to be adjusted. The generated Blob can be directly uploaded to the N9H30 using **XMLHttpRequest** via HTTP PUT or POST, eliminating the need for additional format conversion or compression on the Linux side.

```
/* ③ Convert canvas to Blob (PNG or JPEG) */
function canvasToBlob(canvas, { type = 'image/png', quality = 0.85 } = {}) {
  // `quality` parameter is ignored by PNG but kept for compatibility
  return new Promise(resolve => canvas.toBlob(b => resolve(b), type, quality));
}
```

Finally, after completing the program development, the source code must be compiled to generate the executable file. This executable can then be executed on the NuMaker-HMI-N9H30 development platform. The compilation process in the Linux environment is described below.

Cross-Compilation Workflow in the Linux Environment:

1. Move the *BuildCode* folder from *EC_N9H30_Linux-based Digital Frame System_V1.00* into the N9H30 Linux development environment (**Ubuntu**).

2. When performing cross-compilation, it is important to ensure that the toolchain is correctly configured. In the **Makefile**, the **CC** variable specifies the cross-compiler used to compile the source files listed in **SRCS** into the executable defined by **TARGET**. The **STRIP** command can then be applied to optimize and reduce the size of the generated executable.

```
CC := arm-linux-gcc
STRIP := arm-linux-strip

TARGET = DigitalPhotoFrame
SRCS := DigitalPhotoFrame.c
```

3. Run the following command to load the cross-compiler settings generated by Buildroot. This ensures that subsequent commands such as **gcc** and **make** automatically use the correct toolchain:

```
~/shared/N9H30/buildroot_2024$ source output/host/environment-setup
```

4. Go to the **BuildCode** directory and compile the program using the **make** command:

```
~/shared/N9H30/buildroot_2024/BuildCode$ make
```

To clean the build files or recompile the project, execute the following command:

```
~/shared/N9H30/buildroot_2024/BuildCode$ make clean
```

5. After compilation is completed, place the generated executable file **DigitalPhotoFrame** into *~output/target/opt*, and **repack** the system. The executable can then be run on the NuMaker-HMI-N9H30 development platform.

```
~/shared/N9H30/buildroot_2024$ make linux-rebuild && make
```

- N9H30 System Configuration:
- Before starting **nginx**, the system must first obtain a network IP address.
  The **udhcpc** command requests an IP from the **DHCP** server through the **eth0** interface, and only after acquiring the IP can the Web Server be started.

```
#!/bin/sh

echo q | fbv -i /opt/logo_800x480.png
# 1. Check DHCP (try up to 5 times)
if udhcpc -i eth0 -n 5; then
    echo "DHCP OK – IP acquired."

    # 2. Create required folders for nginx (to avoid 'No such file or directory' errors)
    mkdir -p /var/log/nginx
    mkdir -p /var/cache/nginx/client-body
    mkdir -p /run/nginx

    # 3. Start nginx
    echo "Starting nginx..."
    nginx -t
    nginx &

    # 4. Show current IP address
    IP=$(ifconfig eth0 | awk '/inet /{print $2}')
    echo "Server running at: http://$IP/"

    # 5. Start the Digital Photo Frame application
    ./DigitalPhotoFrame &
else
    #  DHCP failed → show no-network warning image
    echo "DHCP FAILED – showing no network image..."
    fbv /opt/no_network.png
    exit 1
fi
```

- Nginx Configuration:

```
http {
    include       mime.types;
    default_type  application/octet-stream;

    sendfile        on;
    keepalive_timeout  65;

    server {
        listen       80 default_server;
        server_name  _;
```

```
        # Web root for static files (HTML, JS, CSS, etc.)
        root  /opt/www;
        index index.html;

        # Home page: always serve index.html
        location = / {
            try_files /index.html =404;
        }

        # SPA-style routing:
        location / {
            try_files $uri $uri/ /index.html;
        }

        # === Upload folder: support HTTP PUT ================================
        # URL:  /upload/<filename>
        # Path: /opt/photo/<filename>
        location ^~ /photo/ {
        root /opt;

        client_max_body_size 10M;
        dav_methods PUT;
        create_full_put_path on;
        dav_access group:rw all:r;

        autoindex on;
        autoindex_localtime on;
    }
        # ====================================================================

        # Captive portal / OS connectivity checks -> redirect to main page
        location = /hotspot-detect.html       { return 302 /index.html; }
        location = /library/test/success.html { return 302 /index.html; }
        location = /generate_204              { return 302 /index.html; }
        location = /gen_204                   { return 302 /index.html; }
        location = /clients3/generate_204     { return 302 /index.html; }
        location = /connecttest.txt           { return 302 /index.html; }
        location = /ncsi.txt                  { return 302 /index.html; }

        # Simple error page
        error_page 500 502 503 504 /50x.html;
        location = /50x.html { }
    }
}
```

- During the initial setup or when creating the build environment, the system automatically runs the **emWin AppWizard** demo application after booting. Therefore, to launch a custom application instead, the following files must be modified before **pack:**

```
~buildroot_2024/board/nuvoton/nuc9x0/rootfs-emwin/etc/init.d
```

- Add the startup program or script:

```
#!/bin/sh
export TSLIB_CONFFILE=/etc/ts.conf
export TSLIB_TSDEVICE=/dev/input/event0
```

```
export TSLIB_CALIBFILE=/etc/pointercal
export LD_LIBRARY_PATH=/lib/ts/
/sbin/mdev -s
cd opt
./init.sh  &
```

3. Software and Hardware Requirements

3.1 Software Requirements

- Embedded OS: Linux 5.10

3.2 Hardware Requirements

- Circuit components
  - NuMaker-HMI-N9H30

4. Directory Information

The directory structure is shown below.

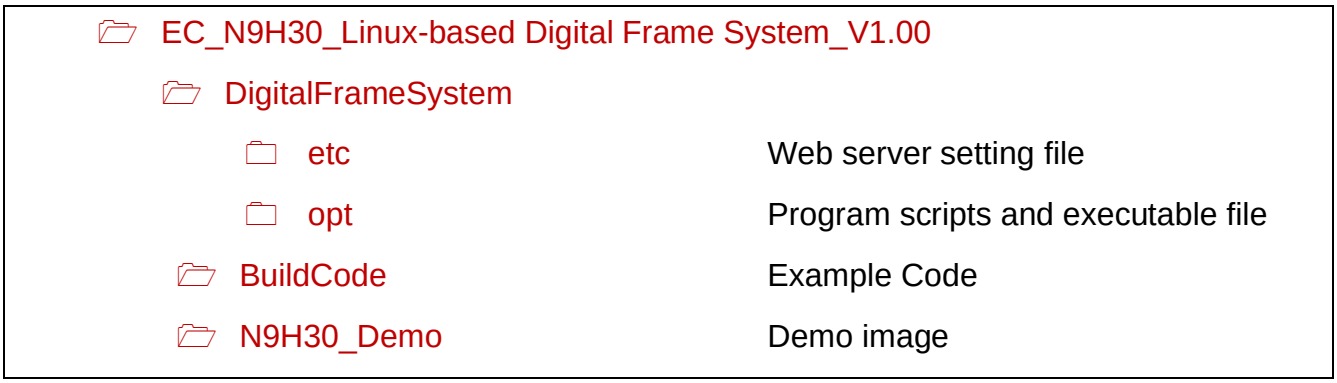

Figure 4-1 Directory Structure

5. Example Code Execution

1. Use the NuWriter tool to program the *n9h30image* file located in the *N9H30_Demo* directory onto the development platform.

   User can load the *n9h30image* file into DDR for this example code.when the *n9h30image* is programmed, the demo application will automatically execute after the system boots up.

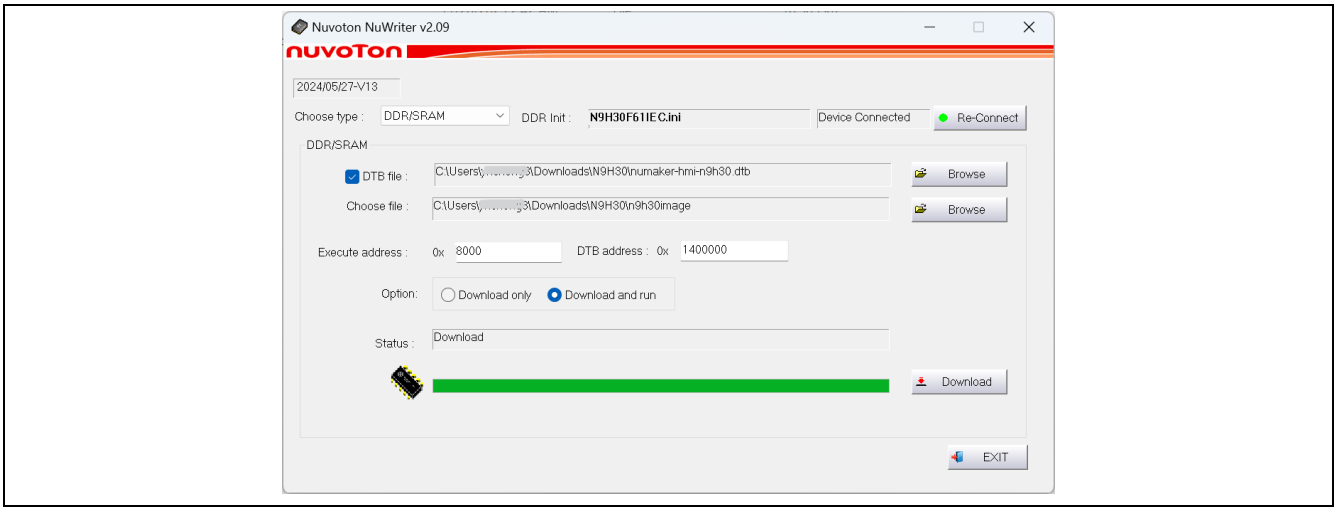

Figure 5-1 NuWriter Tool

2. Connect an Ethernet cable to the **eth0** interface before powering on the NuMaker-HMI-N9H30.

3. Connect the phone and NuMaker-HMI-N9H30 to the same network to upload and update photos.

4. Check the current IP address using **ifconfig.**

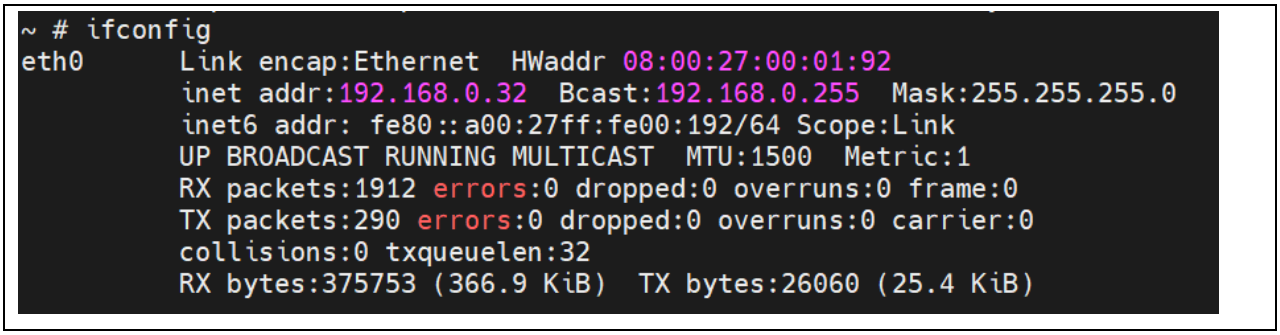

Figure 5-2 IP Check

5. Open the website 192.168.0.32 through the browser.

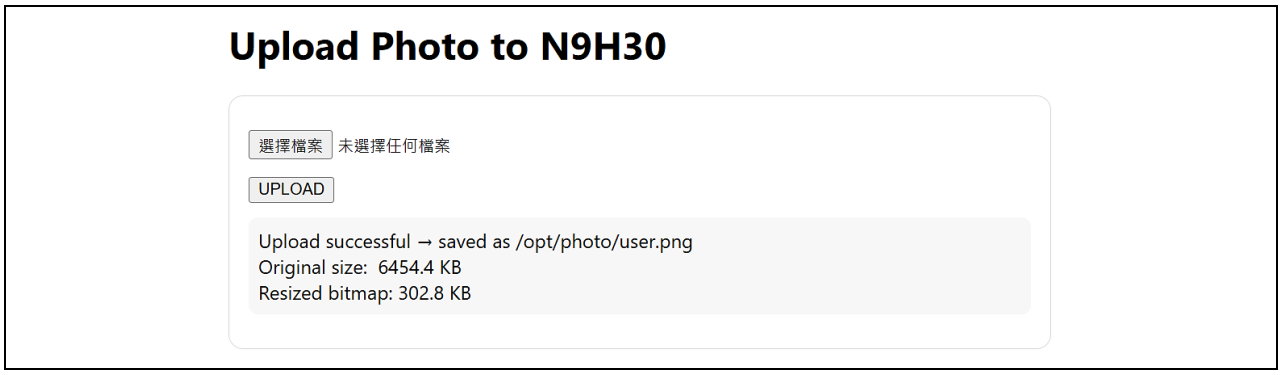

Figure 5-3 Upload Page

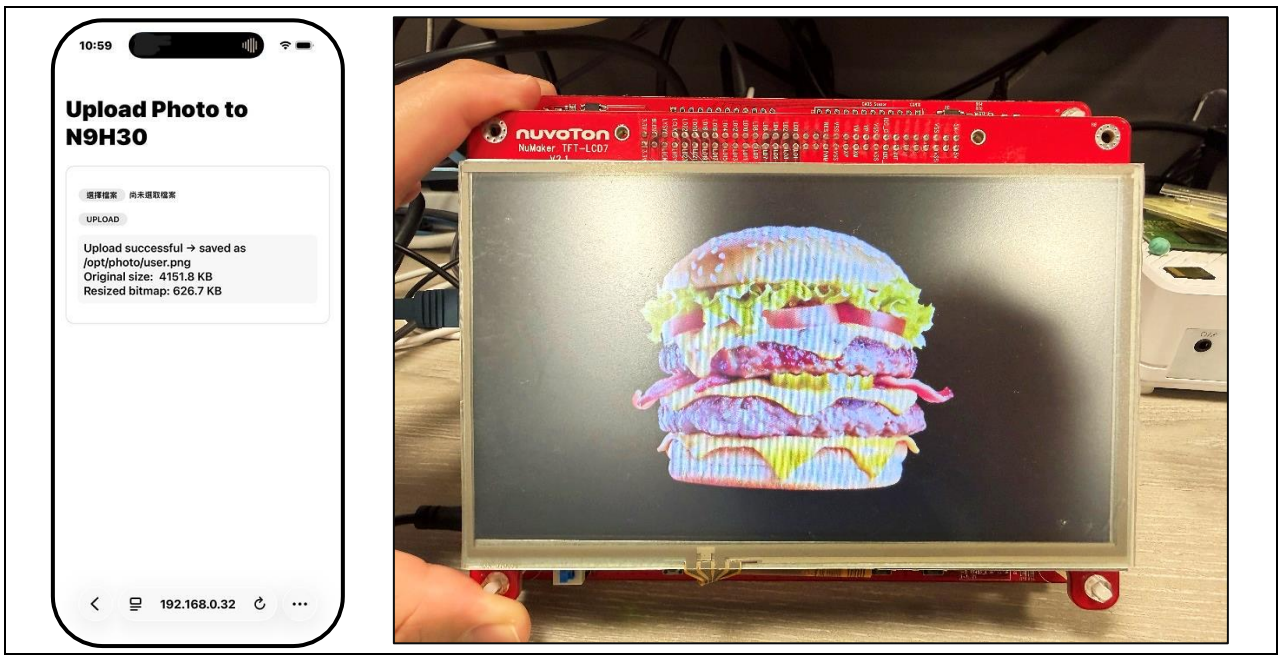

Figure 5-4 Execution Result of Digital Frame System and Mobile Upload Page

Users can compile and deploy the system by following the steps below:

1. Open the N9H30 Linux build environment, then copy the *etc* and *opt* folders from **DigitalFrameSystem** to the corresponding directories under *~output/target*.

   After repacking the image, flash it to the NuMaker-HMI-N9H30 board. This procedure can be referenced in the Code Description section.

2. After entering the system, execute the following steps:

   Go to the *opt* directory:

```
cd opt
```

   Execute the program using the following command:

```
./init.sh  &
```

6. Revision History

Date	Revision	Description
2025.09.10	1.00	Initial version.

Important Notice

Nuvoton Products are neither intended nor warranted for usage in systems or equipment, any malfunction or failure of which may cause loss of human life, bodily injury or severe property damage. Such applications are deemed, "Insecure Usage".

Insecure usage includes, but is not limited to: equipment for surgical implementation, atomic energy control instruments, airplane or spaceship instruments, the control or operation of dynamic, brake or safety systems designed for vehicular use, traffic signal instruments, all types of safety devices, and other applications intended to support or sustain life.

All Insecure Usage shall be made at customer's risk, and in the event that third parties lay claims to Nuvoton as a result of customer's Insecure Usage, customer shall indemnify the damages and liabilities thus incurred by Nuvoton.

Please note that all data and specifications are subject to change without notice.
All the trademarks of products and companies mentioned in this datasheet belong to their respective owners.